

# Nociones básicas de programación

1. [Datos](#)
2. [Instrucciones Elementales](#)
3. [Instrucciones Compuestas](#)
4. [Ejemplos de programas iterativos](#)
5. [Diagramas de Estados](#)
6. [Recursividad](#)
7. ["Dividir para reinar"](#)
8. [Recursividad y Tabulación \(Programación Dinámica\)](#)
9. [Conceptos de Programación Orientada al Objeto \(OOP\)](#)

En esta sección se revisarán los elementos básicos que se van a utilizar para escribir programas. Esto supone que los alumnos ya saben programar, y es sólo un resumen y una ordenación de conceptos. La notación utilizada es la del lenguaje Java, pero los conceptos son más generales y se aplican a muchos otros lenguajes similares.

## Datos

Los programas representan la información que manejan mediante valores llamados "constantes", y dichos valores se almacenan en "variables".

### Variables

```
int k; // entero
float x; // real
double prom; // real de doble precisión
boolean condicion; // verdadero o falso
char c; // un carácter
String nombre; // secuencia de caracteres
```

Nótese que la secuencia `"/"` indica el comienzo de un comentario, el cual se extiende hasta el final de la línea.

### Constantes

```
3 // int
4.50 // float
1e-6 // float
'a' // char
"hola" // String
```

## Instrucciones Elementales

### Asignación

Esta es la instrucción más simple, que permite modificar el valor de una variable:

```
a = E; // asigna a la variable 'a' el valor de la expresión 'E'
```

Ejemplos:

```
k = 0;  
k = k+1;  
k += 1;  
++k;  
k++;
```

Las tres últimas son abreviaturas. La notación "+=" permite evitar repetir el lado izquierdo de la asignación. Las dos últimas incrementan el valor de la variable en 1, pero difieren respecto del valor retornado. En el primer caso (preincremento) se incrementa primero y luego se retorna el valor resultante. En el segundo caso (postincremento) se incrementa después, y se retorna el valor previo de la variable.

## Salida

```
System.out.println(";Hola!");
```

## Instrucciones Compuestas

Estas instrucciones se forman agrupando a otras instrucciones, ya sean elementales o compuestas, usando las reglas de *secuencia*, *alternación* (if) e *iteración* (while).

### Secuencia de instrucciones

Un grupo de instrucciones escritas una a continuación de la otra se ejecutan en ese mismo orden:

```
instrucción1;  
instrucción2;  
. . .
```

También es posible agrupar las instrucciones entre llaves para que sean equivalentes a una sola instrucción:

```
{  
    instrucción1;  
    instrucción2;  
    . . .  
}
```

Ejemplo:

```
// Intercambiar dos valores a y b  
{  
    int aux = a;  
    a = b;  
    b = aux;  
}
```

### Instrucciones condicionales

### Forma 1:

```
if( condición )
    instrucción1;
else
    instrucción2;
```

### Forma 2:

```
if( condición )
    instrucción;
```

Nota: No existe el "endif". Si lo que se desea ejecutar en cada caso es más de una instrucción, hay que escribirlas encerradas entre llaves.

### Ejemplo:

```
// m = max(a,b)
if( a>b )
    m = a;
else
    m = b;
```

### Ejemplo:

```
// a = abs(a)
if( a<0 )
    a = -a; // se omite el "else" si es vacío
```

### Ejemplo:

```
// Ordenar a, b (borrador)
if( a>b )
    intercambiar a, b;
```

La línea destacada no es una instrucción real del lenguaje, es sólo una forma de dejar pendiente esa parte del programa. Más adelante se podrá "refinar" esa pseudo-instrucción definiendo:

```
intercambiar a, b =>
{
    int aux = a;
    a = b;
    b = aux;
}
```

Si se efectúa la sustitución del texto refinado en lugar del que se había escrito originalmente, resulta un texto de programa refinado que cumple con las reglas del lenguaje. Para ayudar a la auto-documentación del programa, se puede conservar la pseudo-instrucción como comentario:

```
// Ordenar a, b
if( a>b )
{ // intercambiar a, b
  int aux = a;
  a = b;
  b = aux;
}
```

Ejemplo: Encontrar el máximo entre un conjunto de variables

```
// m = max(a,b,c)
// Solución 1 (borrador)
if( a>b )
  m = max(a,c) ;
else
  m = max(b,c) ;
```

Realizando las sustituciones respectivas, se obtiene la siguiente versión "refinada":

```
// m = max(a,b,c)
// Solución 1 (versión refinada)
if( a>b )
{
  if( a>c )
    m = a;
  else
    m = c;
}
else
{
  if( b>c )
    m = b;
  else
    m = c;
}
```

Nota: En este caso, las llaves no son realmente necesarias, pero pueden utilizarse si ayudan a la claridad del programa.

Este enfoque de solución tiene la desventaja que es difícil de generalizar. Por ejemplo, el programa que encuentra el máximo de cuatro variables tiene aproximadamente el doble de líneas que éste, y por lo tanto el tamaño del programa va creciendo exponencialmente. Además no hay forma de escribir un programa para un número de variables que no sea conocido *a priori*.

```
// m = max(a,b,c)
// Solución 2 (borrador)
m = a;
m = max(m,b) ;
m = max(m,c) ;
```

Sustituyendo las pseudo-instrucciones, resulta:

```
// m = max(a,b,c)
// Solución 2 (versión refinada)
m = a;
if( b>m )
    m = b;
if( c>m )
    m = c;
```

Con cada instrucción que se ejecuta, el estado del proceso cambia. Para entender lo que está sucediendo en el programa, puede resultar útil intercalar comentarios que describan lo que sabemos que se cumple después de cada instrucción:

```
// m = max(a,b,c)
// Solución 2 (versión refinada y con afirmaciones)
m = a;
// m == max(a)
if( b>m )
    m = b;
// m == max(a,b)
if( c>m )
    m = c;
// m == max(a,b,c)
```

Ese tipo de comentarios se llaman *afirmaciones (assertions)*, y en casos más complejos son fundamentales para entender lo que está sucediendo en un proceso.

La generalización para encontrar el máximo de cuatro o más variables es directa, y en cada caso requiere sólo agregar dos líneas al programa. Más adelante se verá una versión para un número variable de datos.

## Instrucción iterativa

La forma general es:

```
while( condición )
    instrucción;
```

La instrucción se ejecuta en forma reiterada mientras la condición sea verdadera.

Cada vez que se intenta iniciar una nueva iteración (incluyendo la primera vez que ello ocurre) el programa se encuentra en un estado *I* llamado el *invariante* del ciclo.

En general, al escribir un ciclo, se debe establecer la validez inicial del invariante, a través de una *inicialización*. El objetivo del ciclo es llegar a un estado final  $F$ . En cada iteración se debe, además, preservar la validez del invariante.

Ejemplo:

Considere el problema de encontrar el máximo de un número variable de datos, almacenados en un arreglo  $a[1], \dots, a[n]$ . Para verlo en forma iterativa, imagine un proceso en que los datos se van examinando uno a uno, comparándolos con el máximo encontrado hasta el momento. De esta manera, si en un instante dado ya se han examinado los datos hasta  $a[k]$ , entonces se conoce el máximo hasta esa variable.

```
// m = max(a[1], ..., a[n]);
k = 1;
m = a[1];
// m == max(a[1])
// De esta manera trivial se incializa el siguiente invariante:
// Invariante: k<=n && m == max(a[1], ..., a[k])
while( k<n )
{
    ++k;
    // k<=n && m == max(a[1], ..., a[k-1])
    if( a[k]>m )
        m = a[k];
    // k<=n && m == max(a[1], ..., a[k])
}
// m = max(a[1], ..., a[n])
```

Esta última afirmación se deduce del hecho que al terminar el ciclo se sabe que el invariante sigue siendo verdadero, pero la condición del ciclo es falsa. En estricto rigor, la afirmación que podríamos hacer ahí es

```
// k>=n && k<=n && m == max(a[1], ..., a[k])
```

de la cual se deduce la señalada al final del programa.

## ¿Cómo escribir un ciclo?

1. Encontrar un invariante adecuado. Para esto, a menudo es conveniente "relajar" la meta (estado final) al que se desea llegar. Por ejemplo, si se desea obtener:

```
// m == max(a[1], ..., a[n])
```

se puede re-escribir esta condición separándola en dos condiciones que se puedan satisfacer independientemente:

```
// m == max(a[1], ..., a[k]) && k==n
```

Esto, que puede parecer ocioso, es muy útil, porque a continuación se relaja la exigencia de esta condición, haciendo que se cumpla la primera parte, pero dejando que la segunda se satisfaga con " $k \leq n$ ".

2. Escribir la inicialización, la cual debe asegurar que el invariante se cumpla antes de empezar a iterar.
3. Encontrar la condición de término. Esto se obtiene de comparar "qué le falta" al invariante para ser igual al estado final.
4. Escribir el cuerpo del ciclo, el cual debe:
  - conseguir que el proceso avance, de modo que termine algún día, y
  - preservar el invariante.

Estos dos últimos objetivos suelen ser contrapuestos. Al efectuar un avance en el proceso, los valores de las variables cambian, con el resultado que a menudo se deja de satisfacer el invariante. Por lo tanto, el resto del cuerpo del ciclo se suele dedicar a tratar de recuperar la validez del invariante.

## Ejemplos de programas iterativos

### Algoritmos simples de ordenación

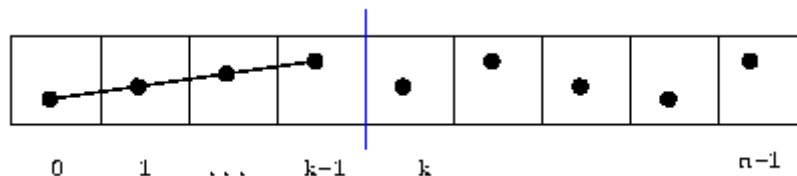
Considere el problema de poner los elementos de un arreglo  $a[0], \dots, a[n-1]$  en orden ascendente.

Se estudiarán varias soluciones, todas ellas consistentes en algoritmos sencillos, pero no muy eficientes. Estas distintas soluciones van a provenir de escoger distintos invariantes, o distintas maneras de preservarlos.

### Ordenación por inserción

Este algoritmo va construyendo un trozo ordenado del arreglo al extremo izquierdo, y en cada iteración le agrega un nuevo elemento a ese grupo.

Invariante:



Esto es: los  $k$  primeros elementos ya están ordenados.

```
// Ordenar  $a[0], \dots, a[k-1]$  por inserción (borrador)
k = 0; // inicialmente no hay elementos ordenados (k=1 también
funcionaría)
while( k < n )
{
    Insertar  $a[k]$  entre  $a[0], \dots, a[k-1]$ ;
    ++k;
}
```

```
}
```

Si la inserción se efectúa correctamente, es evidente que el programa anterior ordena correctamente al conjunto.

El siguiente problema es ver cómo se realiza la inserción:

```
Insertar a[k] entre a[0],...,a[k-1] =>
for( j=k; j>0 && a[j-1]>a[j]; --j )
{
    // intercambiar a[j-1] con a[j]
    t = a[j];
    a[j] = a[j-1];
    a[j-1] = t;
}
```

Al seguir el proceso de la inserción, se puede observar que la variable  $t$  toma siempre el mismo valor: el del elemento que se está insertando. Por lo tanto, se puede optimizar el programa realizando una única asignación a  $t$  antes de comenzar el ciclo. Otra observación es que la mayoría de las asignaciones a  $a[j-1]$  son inútiles, porque esa variable va a ser sobre-escrita en la iteración siguiente. Luego, se puede evitar esas asignaciones, reemplazándolas por una sola al final del ciclo:

```
Insertar a[k] entre a[0],...,a[k-1] =>
// versión optimizada
t = a[k];
for( j=k; j>0 && a[j-1]>t; --j )
    a[j] = a[j-1];
a[j] = t;
```

Efectuando la sustitución de esta versión, se obtiene la siguiente versión final para el algoritmo de ordenación:

```
// Ordenar a[0],...,a[k-1] por inserción
k = 0; // inicialmente no hay elementos ordenados (k=1 también
funcionaría)
while( k<n )
{
    // Insertar a[k] entre a[0],...,a[k-1]
    t = a[k];
    for( j=k; j>0 && a[j-1]>t; --j )
        a[j] = a[j-1];
    a[j] = t;

    ++k;
}
```

El tiempo que demora este algoritmo en el peor caso es del orden de  $n^2$ , lo que se denotará  $O(n^2)$ . Se puede demostrar que esto mismo es cierto si se considera el caso promedio.

## Notación O



$$T(n) = O(f(n)) \Rightarrow T(n) \leq cf(n)$$

$$T(n) = \Omega(g(n)) \Rightarrow T(n) \geq cg(n)$$

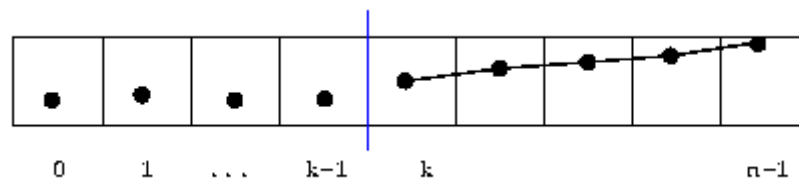
$$T(n) = \Theta(h(n)) \Rightarrow T(n) \text{ es } O(h(n)) \text{ y } \Omega(h(n))$$

$$T(n) = o(p(n)) \Rightarrow T(n) \text{ es } O(p(n)) \text{ y no es } \Theta(p(n))$$

$$c > 0 \text{ constante, para todo } n \geq n_0$$

## Ordenación por Selección

Este algoritmo se basa en hacer pasadas sucesivas sobre los datos. En cada pasada, se encuentra el máximo del arreglo, y se lo lleva al extremo derecho. Una vez hecho esto, ese elemento deja de ser considerado, porque se encuentra ya en su posición definitiva. Esto conduce al siguiente invariante:



En palabras: "Los elementos desde  $k$  hasta  $n-1$  ya están ordenados y son mayores que los primeros  $k$ ".

```
// Ordenar a[0], ..., a[n-1] por selección

k = n; // inicialmente los n primeros están desordenados
while( k >= 2 )
{
    Llevar el max de a[0], ..., a[k-1] hacia a[k-1];
    --k;
}
Donde
Llevar el max de a[0], ..., a[k-1] hacia a[k-1] =>
i = 0; // a[i] es el max hasta el momento
for( j=1; j<=k-1; ++j )
    if( a[j] > a[i] )
        i = j;
// ahora intercambiamos a[i] con a[k-1]
t = a[i];
a[i] = a[k-1];
a[k-1] = t;
```

El tiempo que demora este algoritmo es  $O(n^2)$ , y no hay diferencia entre el peor caso y el caso promedio.

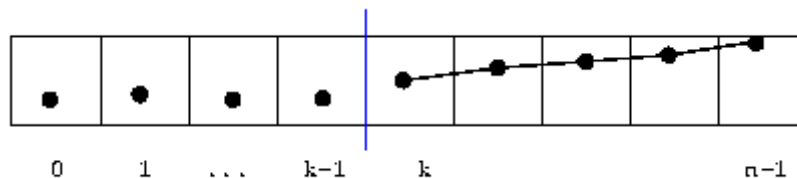
Más adelante se verá una forma diferente de realizar el proceso de encontrar el máximo, que permitirá que ese proceso sea más eficiente. Básicamente, se trata que al encontrar el máximo una vez, es posible obtener información adicional que facilite encontrar luego el segundo máximo, y así sucesivamente.

Una forma de hacer esto es construir un *torneo balanceado*, al estilo de los torneos de tenis. Una vez que se han jugado todos los partidos del torneo, con  $n$  jugadores, si se desea encontrar al (verdadero) sub-campeón, basta con sustituir imaginariamente al campeón por un jugador pésimo, y jugar de nuevo los  $\log n$  partidos en que estuvo involucrado el campeón. El resultado es un método de ordenación que demora tiempo  $O(n \log n)$ .

## Ordenación de la Burbuja

Este método se basa en hacer pasadas de izquierda a derecha sobre los datos, intercambiando pares de elementos adyacentes que estén fuera de orden. Al final de cada pasada, en forma natural el máximo estará en la posición de más a la derecha (que es su posición final) y puede por lo tanto ser excluido en pasadas sucesivas.

Esto conduce al siguiente invariante (idéntico al de ordenación por selección):



El borrador del programa es:

```
// Ordenar a[0], ..., a[n-1] por la burbuja (borrador)
k = n;
while( k>1 )
{
    Hacer una pasada sobre a[0], ..., a[k-1];
    Disminuir k;
}
Donde
Hacer una pasada sobre a[0], ..., a[k-1] =>
for( j=0; j<=k-2; ++j )
    if( a[j]>a[j+1] )
    { // Intercambiar a[j] con a[j+1]
        t = a[j];
        a[j] = a[j+1];
        a[j+1] = t;
    }
y
Disminuir k =>
--k;
```

Esto último puede parecer ocioso, pero pronto se verá que el expresarlo de esta manera da una flexibilidad que resulta útil.

Un problema que presenta este programa es que si el archivo está inicialmente ordenado, el programa igual hace  $n$  pasadas, cuando después de la primera ya podría haberse dado cuenta que el archivo ya estaba ordenado.

Para aprovechar cualquier posible orden que pueda haber en el archivo, se puede hacer que el programa anote ("recuerde") el lugar en donde se produjo el último intercambio. Si la variable  $i$  se define de manera que el último intercambio en una pasada dada fue entre  $a[i-1]$  y  $a[i]$ , entonces todos los elementos desde  $a[i]$  en adelante están ya ordenados (de lo contrario habría habido intercambios más hacia la derecha), y por lo tanto  $k$  se puede disminuir haciendo que sea igual a  $i$ :

```
Hacer una pasada sobre  $a[0], \dots, a[k-1] \Rightarrow$ 
 $i=0;$ 
for(  $j=0; j \leq k-2; ++j$  )
    if(  $a[j] > a[j+1]$  )
        { // Intercambiar  $a[j]$  con  $a[j+1]$ 
           $t = a[j];$ 
           $a[j] = a[j+1];$ 
           $a[j+1] = t;$ 
          //Recordar el lugar del último intercambio
           $i = j+1;$ 
        }
Disminuir  $k \Rightarrow$ 
 $k=i;$ 
```

El tiempo que demora este algoritmo tanto en el peor caso como en promedio es  $O(n^2)$ .

## Cálculo de $x^n$

Un algoritmo simple consiste en multiplicar  $n$  veces:

```
// Algoritmo simple
 $y = 1;$ 
for(  $j=n; j>0; --j$  )
     $y = y*x;$ 
```

Este algoritmo evidentemente toma tiempo  $O(n)$ , y su invariante se puede escribir como

$$y * x^j == x^n$$

Es posible encontrar un algoritmo sustancialmente más eficiente de la siguiente manera. Primero se desvinculan las dos ocurrencias de  $x$  en el invariante:

```
 $y = 1;$ 
 $z = x;$ 
for(  $j=n; j>0; --j$  )
     $y = y*z;$ 
```

con invariante

$$y * z^j == x^n$$

Esto podría parecer ocioso, pero permite hacer una optimización al observar que está permitido modificar la variable  $z$  al inicio del ciclo siempre que se mantenga la validez del invariante. En particular, si  $j$  resulta ser *par*, podemos elevar  $z$  al cuadrado si al mismo tiempo dividimos  $j$  por 2. De esta manera, el invariante sigue igual, pero  $j$  disminuye mucho más rápido.

Mejor todavía, si esto se puede hacer una vez, entonces se puede hacer muchas veces siempre que  $j$  siga siendo par:

```
y = 1;
z = x;
for( j=n; j>0; --j ) // Invariante: y * zj == xn
{
    while( j es par )
    {
        z = z*z;
        j = j/2;
    }
    y = y*z;
}
```

La detección que  $j$  es par se puede implementar como

```
j es par =>
j&1 == 0
```

Este algoritmo demora tiempo  $O(\log n)$ , lo cual se debe a que  $j$  sólo se puede dividir  $\log n$  veces por 2 antes de llegar a 1. Es cierto que  $j$  sólo se divide cuando es par, pero si es impar en una iteración del `for`, está garantizado que será par a la siguiente.

## Diagramas de Estados

Un diagrama de estados nos permite visualizar los diferentes estados por los que va pasando un programa. Las transiciones de un estado a otro se realizan ya sea incondicionalmente o bajo una condición. Además, pueden ir acompañadas de una acción que se realiza junto con la transición.

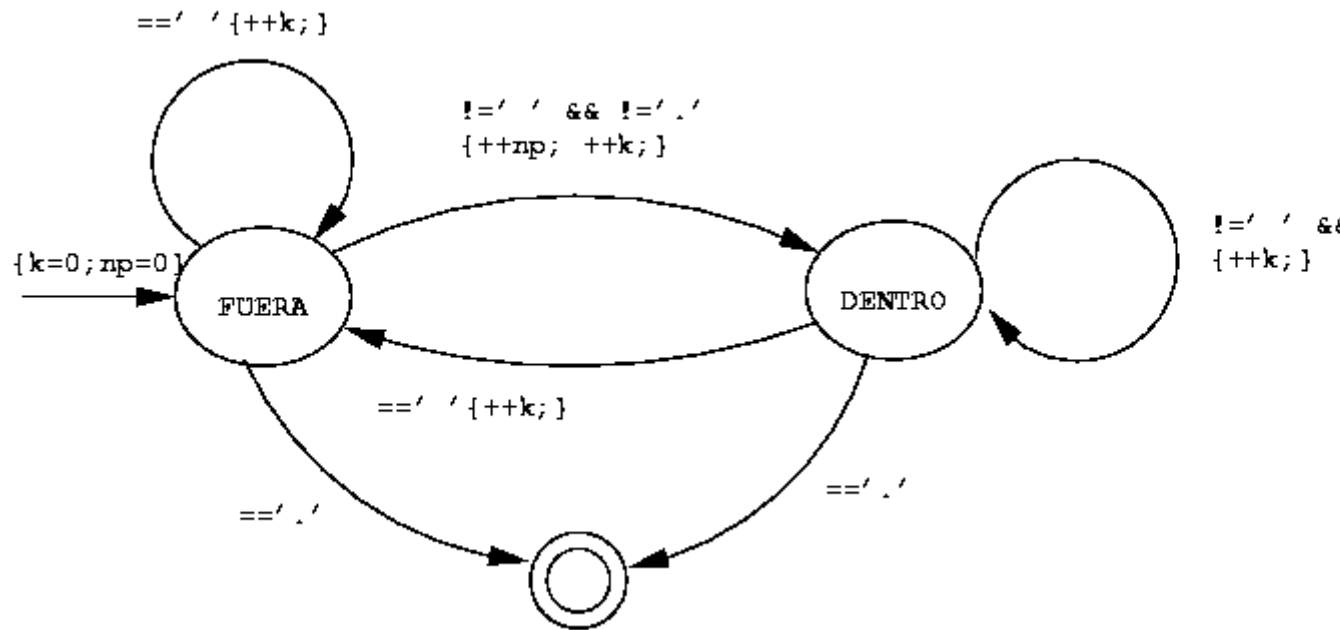
*Ejemplo:* Contar palabras en una frase.

Para simplificar, supongamos que la frase está almacenada en un string  $s$ , y supongamos que la frase termina con un punto. Por ejemplo,

```
String s = "Este es un ejemplo.";
```

Para los fines de este ejemplo, diremos que una "palabra" es cualquier secuencia de caracteres consecutivos distintos de blanco (y punto).

Para resolver este problema, examinaremos los caracteres del string de izquierda a derecha, usando `charAt(k)`, y lo que se haga con cada caracter depende si estábamos dentro o fuera de una palabra. Esto último corresponde al estado del programa.



Veremos dos formas distintas de llevar este diagrama de transición a un programa. A pesar que ambos se ven muy distintos, en realidad representan exactamente el mismo proceso:

// Version 1

```

np = 0;
estado = FUERA;
for( k=0; (c=s.charAt(k))!='.'; ++k )
{
    if( estado==FUERA )
    {
        if( c!=' ' )
        {
            ++np;
            estado = DENTRO;
        }
    }
    else // estado==DENTRO
        if( c==' ' )
            estado = FUERA;
}

```

// Version 2

```

k = 0;
np = 0;
while( s.charAt(k)!='.' )
{ // estado==FUERA
    while( s.charAt(k)==' ' )
        ++k;
    if( s.charAt(k)=='.' )
        break;
    ++np;
    ++k;

    // estado==DENTRO
}

```

```

while( s.charAt(k)!=' ' && s.charAt(k)!='.' )
    ++k;
if( s.charAt(k)=='.' )
    break;
++k;
}

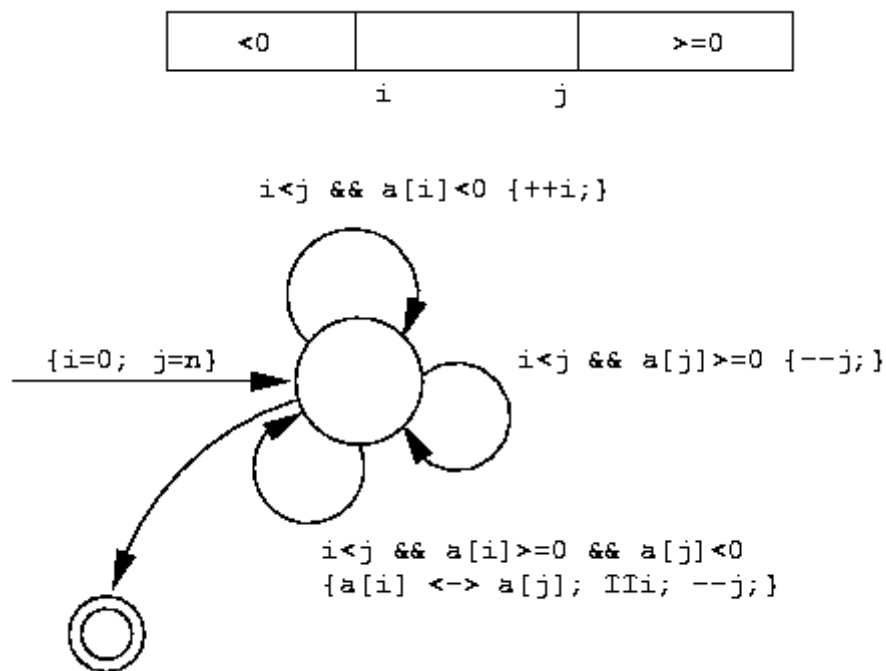
```

## Problema

Reordenar los elementos de  $a[0], \dots, a[n]$  dejando a la izquierda los  $<0$  y a la derecha los  $\geq 0$ .

*Solución 1:*

Invariante:



// Version 1

```

i = 0;
j = n;
while( i < j )
{
    if( a[i] < 0 )
        ++i;
    else if( a[j] >= 0 )
        --j;
    else
    {
        a[i] <-> a[j];
        ++i;
        --j;
    }
}

```

// Version 2

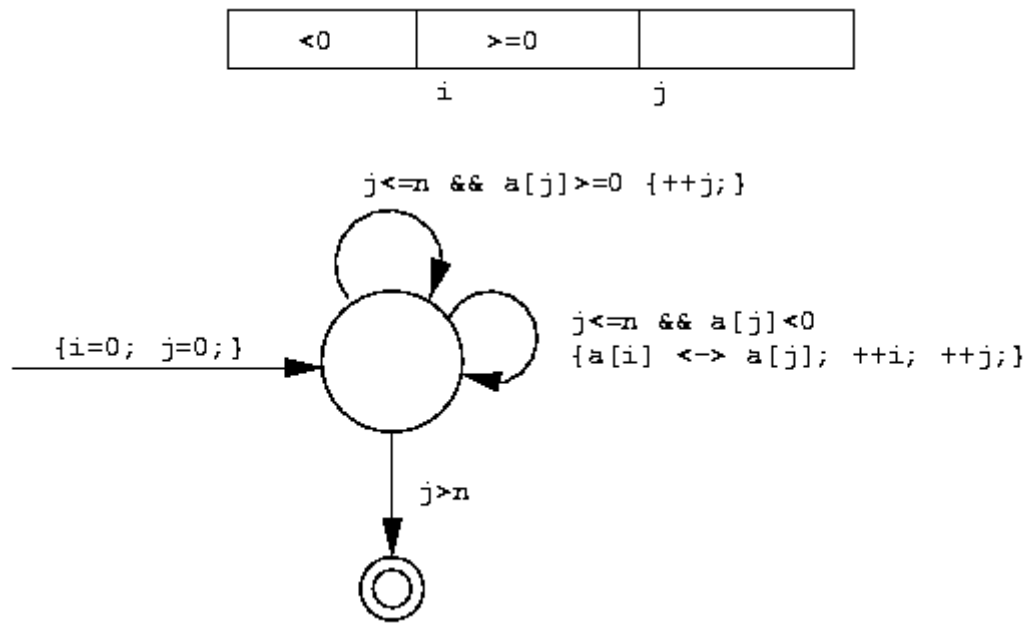
```

i = 0;
j = n;
while( i < j )
{
    while( i < j && a[i] < 0 )
        ++i;
    while( i < j && a[j] >= 0 )
        --j;
    if( i < j )
    {
        a[i] <-> a[j];
        ++i;
        --j;
    }
}

```

*Solución 2:*

Invariante:



```
i = 0;
for( j=0; j<=n; ++j )
    if( a[j]<0 )
    {
        a[i] <-> a[j];
        ++i;
    }
```

## Recursividad

Al programar en forma recursiva, buscamos dentro de un problema otro subproblema que posea su misma estructura.

*Ejemplo:* Calcular  $x^n$ .

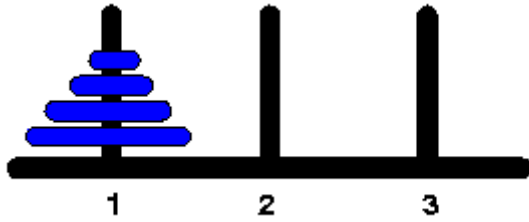
// Version 1

```
public static float elevar( float x, int n )
{
    if( n==0 )
        return 1;
    else
        return x * elevar(x, n-1);
}
```

// Version 2

```
public static float elevar( float x, int n )
{
    if( n==0 )
        return 1;
    elseif( n es impar )
        return x * elevar( x, n-1 );
    else
        return elevar( x*x, n/2 );
}
```

*Ejemplo:* Torres de Hanoi.



```
public class TorresDeHanoi{
    static void Hanoi( int n, int a, int b, int c ){
        if (n>0){
            Hanoi( n-1, a, c, b );
            System.out.println( a + " --> " + c );
            Hanoi( n-1, b, a, c );
        }
    }
    public static void main(String[] args){
        Hanoi(Integer.parseInt(args[0]), 1, 2, 3);
    }
}
```

## "Dividir para reinar"

Este es un método de diseño de algoritmos que se basa en subdividir el problema en sub-problemas, resolverlos recursivamente, y luego combinar las soluciones de los sub-problemas para construir la solución del problema original.

*Ejemplo:* Multiplicación de Polinomios.

Supongamos que tenemos dos polinomios con  $n$  coeficientes, o sea, de grado  $n-1$ :

$$A(x) = a_0 + a_1x + \dots + a_{n-1}x^{n-1}$$

$$B(x) = b_0 + b_1x + \dots + b_{n-1}x^{n-1}$$

representados por arreglos  $a[0], \dots, a[n-1]$  y  $b[0], \dots, b[n-1]$ . Queremos calcular los coeficientes del polinomio  $C(x)$  tal que  $C(x) = A(x) * B(x)$ .

Un algoritmo simple para calcular esto es:

```
// Multiplicación de polinomios

for( k=0; k<=2*n-2; ++k )
    c[k] = 0;
for( i=0; i<n; ++i)
    for( j=0; j<n; ++j)
        c[i+j] += a[i]*b[j];
```

Evidentemente, este algoritmo requiere tiempo  $O(n^2)$ . ¿Se puede hacer más rápido?

Supongamos que  $n$  es par, y dividamos los polinomios en dos partes. Por ejemplo, si



$$A(x) = 2 + 3x - 6x^2 + x^3$$

entonces se puede reescribir como

$$A(x) = (2+3x) + (-6+x)x^2$$

y en general

$$\begin{aligned} A(x) &= A'(x) + A''(x) * x^{n/2} \\ B(x) &= B'(x) + B''(x) * x^{n/2} \end{aligned}$$

Entonces

$$\begin{aligned} C &= (A' + A''x^{n/2}) * (B' + B''x^{n/2}) \\ &= A'B' + (A'B'' + A''B') * x^{n/2} + A''B'' * x^n \end{aligned}$$

Esto se puede implementar con 4 multiplicaciones recursivas, cada una involucrando polinomios de la mitad del tamaño que el polinomio original. Si llamamos  $T(n)$  al número total de operaciones, éste obedece la ecuación de recurrencia

$$T(n) = 4 * T(n/2) + K * n$$

donde  $K$  es alguna constante cuyo valor exacto no es importante.

*Teorema*

Las ecuaciones de la forma

$$T(n) = p * T(n/q) + K * n$$

con tienen solución

$$\begin{aligned} T(n) &= O(n^{\log_q p}) & (p > q) \\ T(n) &= O(n) & (p < q) \\ T(n) &= O(n \log n) & (p = q) \end{aligned}$$

Por lo tanto la solución del problema planteado ( $p=4$ ,  $q=2$ ) es

$$T(n) = O(n^{\log_2 4}) = O(n^2)$$

lo cual no mejora al algoritmo visto inicialmente.

Pero... hay una forma más eficiente de calcular  $C(x)$ . Si calculamos:

$$\begin{aligned} D &= (A' + A'') * (B' + B'') \\ E &= A' * B' \\ F &= A'' * B'' \end{aligned}$$

entonces

$$C = E + (D - E - F) * x^{n/2} + F * x^n$$

Lo cual utiliza sólo 3 multiplicaciones recursivas, en lugar de 4. Esto implica que

$$T(n) = O(n^{\log_2 3}) = O(n^{1.59})$$

## Recursividad y Tabulación (Programación Dinámica)

A veces la simple recursividad no es eficiente.

*Ejemplo:* Números de Fibonacci.

Los números de Fibonacci se definen mediante la recurrencia

$$\begin{aligned} f_n &= f_{n-1} + f_{n-2} \quad (n \geq 2) \\ f_0 &= 0 \\ f_1 &= 1 \end{aligned}$$

cuyos primeros valores son

n	0	1	2	3	4	5	6	7	8	9	10	11	.	.	.
f <sub>n</sub>	0	1	1	2	3	5	8	13	21	34	55	89	.	.	.

Se puede demostrar que los números de Fibonacci crecen exponencialmente, como una función  $O(\varphi^n)$  donde  $\varphi = 1.618\dots$

El problema que se desea resolver es calcular  $f_n$  para un  $n$  dado.

La definición de la recurrencia conduce inmediatamente a una solución recursiva:

```
public static int F( int n )
{
    if( n <= 1 )
        return n;
    else
        return F(n-1)+F(n-2);
}
```

Lamentablemente, este método resulta muy ineficiente. En efecto, si llamamos  $T(n)$  al número de operaciones de suma ejecutadas para calcular  $f_n$ , tenemos que

$$\begin{aligned} T(0) &= 0 \\ T(1) &= 0 \\ T(n) &= 1 + T(n-1) + T(n-2) \end{aligned}$$

La siguiente tabla muestra los valores de  $T(n)$  para valores pequeños de  $n$ :

n	0	1	2	3	4	5	6	7	8	9	10	...
T(n)	0	0	1	2	4	7	12	20	33	54	88	...

*Ejercicio:* Demostrar que  $T(n) = f_{n+1} - 1$ .

Por lo tanto, el tiempo que demora el cálculo de  $F(n)$  crece *exponencialmente* con  $n$ , lo cual hace que este método sea inútil excepto para valores muy pequeños de  $n$ .

El origen de esta ineficiencia es que la recursividad calcula una y otra vez los mismos valores, porque no guarda memoria de haberlos calculado antes.

Una forma de evitarlo es utilizar un arreglo auxiliar `fib[]`, para anotar los valores ya calculados. Un método general es inicializar los elementos de `fib` con algún valor especial "nulo". Al llamar a  $F(n)$ , primero se consulta el valor de `fib[n]`. Si éste no es "nulo", se retorna el valor almacenado en el arreglo. En caso contrario, se hace el cálculo recursivo y luego se anota en `fib[n]` el resultado, antes de retornarlo. De esta manera, se asegura que cada valor será calculado recursivamente sólo una vez.

En casos particulares, es posible organizar el cálculo de los valores de modo de poder ir llenando el arreglo en un orden tal que, al llegar a `fib[n]`, ya está garantizado que los valores que se necesitan (`fib[n-1]` y `fib[n-2]`) ya hayan sido llenados previamente. En este caso, esto es muy sencillo, y se logra simplemente llenando el arreglo en orden ascendente de subíndices:

```
fib[0] = 0;
fib[1] = 1;
for( j=2; j<=n; ++j )
    fib[j] = fib[j-1]+fib[j-2];
```

El tiempo total que esto demora es  $O(n)$ .

Esta idea se llama *programación dinámica* cuando se la utiliza para resolver problemas de optimización, y veremos algunas aplicaciones importantes de ella a lo largo del curso.

¿Es posible calcular  $f_n$  más rápido que  $O(n)$ ? Si bien podría parecer que para calcular  $f_n$  sería necesario haber calculado todos los valores anteriores, esto no es cierto, y existe un método mucho más eficiente.

Tenemos

$$\begin{aligned}f_n &= f_{n-1} + f_{n-2} \\ f_0 &= 0 \\ f_1 &= 1\end{aligned}$$

Esta es una ecuación de recurrencia de segundo orden, porque  $f_n$  depende de los dos valores inmediatamente anteriores. Definamos una función auxiliar

$$g_n = f_{n-1}$$

Con esto, podemos re-escribir la ecuación para  $f_n$  como un sistema de dos ecuaciones de *primer* orden:

$$\begin{aligned}f_n &= f_{n-1} + g_{n-1} \\ g_n &= f_{n-1} \\ f_1 &= 1 \\ g_1 &= 0\end{aligned}$$

Lo anterior se puede escribir como la ecuación vectorial

$$\mathbf{f}_n = \mathbf{A} * \mathbf{f}_{n-1}$$

donde

$$\mathbf{f}_n = \begin{bmatrix} f_n \\ g_n \end{bmatrix} \quad \mathbf{A} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

con la condición inicial

$$\mathbf{f}_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

La solución de esta ecuación es

$$\mathbf{f}_n = \mathbf{A}^{n-1} * \mathbf{f}_1$$

lo cual puede calcularse en tiempo  $O(\log n)$  usando el método rápido de elevación a potencia visto anteriormente.

## Conceptos de Programación Orientada al Objeto (OOP)

Un *objeto* combina *datos* y *operaciones* (métodos).

El principio básico es el de *encapsulamiento* (ocultamiento de información). Esto permite separar el "qué" (especificación funcional, pública) del "cómo" (implementación, privada).

Conceptos asociados a la programación orientada a objetos, para apoyar la reutilización de código:

- Código genérico: la lógica de un algoritmo debe poder escribirse independientemente del tipo de los datos.
- Herencia: permite extender la funcionalidad de un objeto.
- Polimorfismo: permite que se seleccione automáticamente la operación apropiada según el tipo y número de los parámetros.

### Clases

En Java un objeto es una *instancia* de una clase.

*Ejemplo:*

```
// Clase Entero, que permite leer y
// guardar un valor en una variable entera

public class Entero
{
```

```

        // Datos privados
        private int valor;

        // Métodos públicos
        public int leer()
        {
            return valor;
        }
        public void guardar( int x )
        {
            valor = x;
        }
    }

// Ejemplo de programa principal

public class Prueba
{
    public static void main( String[] args )
    {
        Entero m = new Entero();

        m.guardar( 5 );
        System.out.println( "m=" + m.leer() );
    }
}

```

## **Tipos de métodos**

### **Constructores**

Permiten inicializar el objeto. Puede haber varios constructores con distinto número y tipos de parámetros.

Si no hay un constructor definido, los campos se inicializan automáticamente con valores nulos.

El constructor debe tener el mismo nombre que la clase.

*Ejemplo:* Clase para almacenar fechas:

```

public class Fecha
{
    private int a;
    private int m;
    private int d;

    // Constructor con parámetros
    public Fecha( int aa, int mm, int dd )
    {
        a = aa;
        m = mm;
        d = dd;
    }

    // Constructor sin parámetros
    public Fecha()
    {

```

```

        a = 2001;
        m = 1;
        d = 1;
    }
}

```

*Ejemplos de uso:*

```

Fecha f1 = new Fecha();
Fecha f2 = new Fecha( 2001, 4, 11 );

```

## "Mutators" y "accessors"

Las variables de una clase típicamente son privadas. Para mirar su valor, o para modificarlo, hay que utilizar métodos *ad hoc* (como `leer` y `guardar` en el ejemplo de la clase `Entero`).

Esto es un mayor grado de burocracia, pero aísla a los usuarios de una clase de los detalles de implementación de ella, y evita que se vean afectados por eventuales cambios en dicha implementación.

## toString

Al imprimir un objeto `a` usando `println`, automáticamente se invoca a

```
a.toString()
```

para convertirlo a una forma imprimible. Esto mismo ocurre cada vez que se utiliza `a` en un contexto de `String`.

En el ejemplo, si vamos a imprimir objetos de tipo `Fecha`, debemos proveer una implementación de `toString` dentro de esa clase:

```

public String toString()
{
    return d + "/" + m + "/" + a;
}

```

## equals

El método `equals` se utiliza para ver si dos objetos tienen el mismo valor. Se invoca

```
if( x.equals(y) )
```

y se declara como

```
public boolean equals( Object b )
```

El tipo `Object` usado aquí es un tipo de objeto "universal" del cual se derivan todos los otros. El siguiente ejemplo muestra una implementación de `equals` para la clase `Fecha`:

```
public boolean equals( Object b )
```

```

{
    if( !(b instanceof Fecha) )
        return false; // el otro objeto no era de tipo Fecha

    Fecha f = (Fecha) b; // para verlo como una Fecha

    return a==f.a && m==f.m && d==f.d;
}

```

## **this**

La referencia `this` identifica al objeto actual. Permite desde dentro de la clase acceder los campos propios diciendo, por ejemplo, `this.a`. Esto en realidad es redundante, porque significa lo mismo que decir simplemente `a`, pero puede ser más claro en la lectura.

También permite comparar si este objeto es el *mismo* que otro (no sólo si tienen el mismo contenido, sino si ambas referencias apuntan al mismo objeto).

El otro uso de `this` es como constructor, para llamar a otro constructor de la misma clase. Por ejemplo, el constructor sin parámetros de la clase `Fecha` podría haber sido declarado como:

```

public Fecha()
{
    this( 2001, 1, 1 );
}

```

## **Campos estáticos**

Hay dos tipos de campos estáticos:

```

public final static double PI = 3.14159; // constante

private static int precioActual = 1300; // variable compartida
// por todos los objetos de esa clase

```

## **Métodos estáticos**

Los métodos estáticos están asociados a una *clase*, no a objetos particulares dentro de ella.

*Ejemplos:*

```

Math.sin
Integer.parseInt

```

Los métodos estáticos no pueden hacer uso de la referencia `this`.

## **Packages**

Las clases se pueden agrupar en "paquetes". Para esto, cada clase debe precederse de

```
package P;
class C
{
    . . .
}
```

Cuando a una variable no se le pone `public` ni `private`, es visible sólo dentro del mismo package.

La clase `C` se denomina `P.C`, pero si antes decimos `import P.C;` o bien `import P.*;`, entonces podemos referirnos a la clase simplemente como `C`;

## Herencia

Principio que permite reutilizar trabajo ya hecho. Se basa en la relación **is-a**.

*Ejemplo:*

Círculo **is-a** Figura

Auto **is-a** Vehículo

Las clases forman una jerarquía en base a la relación de herencia.

Otro tipo de relación distinta es **has-a**. Por ejemplo:

Auto **has-a** Manubrio

Este tipo de relación se llama *agregación* y a menudo es más importante que la herencia.

Clase base:

La clase de la cual se derivan otras

Clase derivada:

Hereda todas las propiedades de la clase base. Luego puede agregar campos y métodos, o redefinir métodos.

Los cambios que se hagan en la clase derivada *no* afectan a la clase base.

*Sintaxis:*

```
public class Derivada extends Base
{
    . . .
}
```

- Los campos adicionales generalmente son privados.
- Los métodos de la clase base que no se redefinen en la clase derivada se heredan sin cambio, excepto por el constructor.
- Los métodos que se redefinen tienen prioridad.
- Se pueden agregar nuevos métodos.
- Los métodos públicos no se pueden redefinir como privados.



## Visibilidad

Los campos privados de la clase base *no* se ven desde las clases derivadas. Para que un campo de este tipo sea visible debe declararse como `protected`. Esta posibilidad debe usarse con mucho cuidado.

## Constructores

Cada clase define su propio constructor, el cual lleva el mismo nombre que la clase.

Si no se define un constructor, se genera automáticamente un constructor sin parámetros que:

- llama al constructor con cero parámetros de la clase base, para la parte heredada, y luego
- inicializa con valores nulos los campos restantes.

Si se escribe un constructor, su primera instrucción puede ser una llamada a

```
super( ... );
```

lo cual invoca al constructor de la clase base.

## final

Si un método se declara como `final`, entonces no puede ser redefinido en las clases derivadas.

Análogamente, una clase declarada como `final` no puede ser extendida.

## Métodos abstractos

Un método abstracto declara funcionalidad, pero *no* la implementa. Las clases derivadas deben proveer implementaciones para estos métodos.

Una clase abstracta es una clase que tiene al menos un método abstracto. No se puede crear objetos pertenecientes a una clase abstracta (no se puede ejecutar `new`).

*Ejemplo:*

```
abstract class Figura
{
    private String nombre;

    abstract public double area();

    public Figura( String n )
    {
        nombre = n;
    }
    // Este constructor no puede ser invocado directamente,
    // sólo lo usan las clases derivadas
```

```

        final public double compArea( Figura b )
        {
            return area() - b.area();
        }

        final public String toString()
        {
            return nombre + " de area " + area();
        }
    }

// Clases derivadas

public class Circulo extends Figura
{
    static final private double PI = 3.141592653;
    private double radio;

    public Circulo( double r )
    {
        super( "circulo" );
        radio = r;
    }
    public double area()
    {
        return PI*radio*radio;
    }
}

public class Rectangulo extends Figura
{
    private double largo;
    private double ancho;

    public Rectangulo( double l, double a )
    {
        super( "rectangulo" );
        largo = l;
        ancho = a;
    }
    public double area()
    {
        return largo*ancho;
    }
}

public class Cuadrado extends Rectangulo
{
    public Cuadrado( double lado )
    {
        super( lado, lado );
    }
}

```

## Herencia múltiple

En algunos lenguajes, una clase puede heredar de más de una clase base. En Java esto no se permite, lo cual evita los conflictos que se podrían producir al heredarse definiciones incompatibles de métodos y variables.

## Interfaz

Una interfaz es un mecanismo que permite lograr algunos de los efectos de la herencia múltiple, sin sus problemas.

Una interfaz es una clase que sólo tiene métodos públicos abstractos y campos públicos estáticos finales.

Se dice que una clase *implementa* a la interfaz si provee definiciones para todos los métodos abstractos de la interfaz.

Una clase puede extender sólo a una clase base, pero puede implementar muchas interfaces.

*Ejemplo:*

```
package Definiciones;
public interface Comparable
{
    public int Compare( Comparable b );
}

final public class Entero implements Comparable
{
    private int valor;

    public Entero( int x )
    {
        valor = x;
    }
    public String toString()
    {
        return Integer.toString( valor );
    }
    public int valorEntero()
    {
        return valor;
    }
    public int Compare( Comparable b )
    {
        return valor - ((Entero) b).valor;
    }
}
```

## Uso de interfaces para implementar componentes genéricas

*Ejemplo:* Ordenación por inserción que ordena objetos de cualquier tipo.

```
package Ordenacion;
import Definiciones.*;

public static void insercion( Comparable[] a )
{
    for( int k=0; k<a.length; ++k )
    {
        int j;
```

```

        Comparable t = a[k];

        for( j=k; j>0 &&
            t.Compare(a[j-1])<0; --j)
            a[j] = a[j-1];
        a[j] = t;
    }
}

```

***Ejemplo de uso:*** Ordenar enteros entregados como argumentos.

```

import Definiciones.*;

public class PruebaOrdenacion
{
    public static void main( String[] args )
    {
        Entero[] a = new Entero[args.length];

        for( int i=0; i<args.length; ++i )
            a[i] = new Entero( Integer.parseInt(args[i]) );

        Ordenacion.insercion( a );

        for( int i=0; i<a.length; ++i )
            System.out.println( a[i] );
    }
}

```

# Estructuras de datos básicas

1. [Arreglos.](#)
2. [Punteros y variables de referencia.](#)
3. [Listas enlazadas.](#)
4. [Árboles.](#)
  - [Árboles binarios.](#)
  - [Árboles generales.](#)

Toda la información que se maneja dentro de un computador se encuentra almacenada en su memoria, que en términos simples es una secuencia de caracteres (bytes) en donde se encuentran las instrucciones y datos a los que se accede directamente a través del procesador del computador.

Los sistemas o métodos de organización de datos que permiten un almacenamiento eficiente de la información en la memoria del computador son conocidos como *estructuras de datos*. Estos métodos de organización constituyen las piezas básicas para la construcción de algoritmos complejos, y permiten implementarlos de manera eficiente.

En el presente capítulo se presentan las estructuras de datos básicas como son arreglos, listas enlazadas y árboles, con las cuales se implementarán posteriormente los *tipos de datos abstractos*.

## Arreglos

Un *arreglo* es una secuencia contigua de un número fijo de elementos homogéneos. En la siguiente figura se muestra un arreglo de enteros con 10 elementos:

<i>índice</i>	0	1	2	3	4	5	6	7	8	9
<i>elementos</i>	80	45	2	21	92	17	5	65	14	34

En Java un arreglo se define como:

```
tipo[] nombre = new tipo[n_elem];
```

donde *tipo* corresponde al tipo de los elementos que contendrá el arreglo (enteros, reales, caracteres, etc..), *nombre* corresponde al nombre con el cual se denominará el arreglo, y *n\_elem* corresponde al número de elementos que tendrá el arreglo. Para el caso del ejemplo presentado, la declaración del arreglo de enteros es:

```
int[] arreglo = new int[10];
```

Para acceder a un elemento del arreglo se utiliza un *índice* que identifica a cada elemento de manera única. Los índices en Java son números enteros correlativos y comienzan desde cero, por lo tanto, si el arreglo contiene *n\_elem* elementos el índice del

último elemento del arreglo es  $n_{elem}-1$ . El siguiente código muestra como se puede inicializar el arreglo del ejemplo, luego de ser declarado:

```
arreglo[0]=80; //el primer indice de los arreglos en Java es 0
arreglo[1]=45;
arreglo[2]=2;
arreglo[3]=21;
arreglo[4]=92;
arreglo[5]=17;
arreglo[6]=5;
arreglo[7]=65;
arreglo[8]=14;
arreglo[9]=34; //el ultimo indice del arreglo es 10-1 = 9
```

También se puede declarar e inicializar el arreglo en una sola línea:

```
int[] arreglo={80, 45, 2, 21, 92, 17, 5, 65, 14, 34};
```

Una ventaja que tienen los arreglos es que el costo de acceso de un elemento del arreglo es constante, es decir no hay diferencias de costo entre acceder el primer, el último o cualquier elemento del arreglo, lo cual es muy eficiente. La desventaja es que es necesario definir a priori el tamaño del arreglo, lo cual puede generar mucha pérdida de espacio en memoria si se definen arreglos muy grandes para contener conjuntos pequeños de elementos (Nota: en Java es posible hacer crecer el tamaño de un arreglo de manera dinámica).

## Punteros y variables de referencia

Un *puntero* es una variable que almacena la dirección de memoria de otra variable, es decir, almacena el valor del lugar físico en la memoria en donde se encuentra almacenada dicha variable. Si se imagina que la memoria del computador es un gran arreglo de bytes, la dirección de memoria correspondería al índice de los casilleros de dicho arreglo, que es precisamente lo que se almacena en el puntero.

En algunos lenguajes de programación, por ejemplo C, es posible declarar explícitamente punteros para distintos tipos de variables, e incluso es posible realizar aritmética de punteros para realizar operaciones de manera muy eficiente, a cambio de "oscurecer" el código del programa y con una alta probabilidad de cometer errores de programación difíciles de detectar.

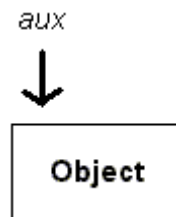
En Java no se puede declarar punteros de manera explícita ni tampoco realizar aritmética de punteros. Por lo tanto es imposible en Java tener un puntero a cualquiera de los tipos primitivos: enteros, reales, caracteres y booleanos. Los *strings* y arreglos *no son tipos primitivos en Java*.

Una *variable de referencia*, o simplemente una *referencia*, es una variable que almacena la dirección de memoria en donde se ubica un *objeto*. Nótese que si bien la definición es prácticamente idéntica a la de puntero, la diferencia radica en que una referencia sólo puede apuntar a *objetos* residentes en memoria, lo cual excluye a los tipos primitivos. A partir de esta definición se puede concluir que toda variable en Java, que no sea de tipo primitivo, es una *referencia*.

Por ejemplo, todas las clases en Java heredan de la clase *Object*. Una instancia de esta clase se declara como:

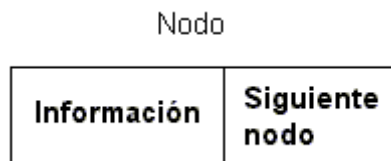
```
Object aux=new Object();
```

La variable *aux* es una referencia a un objeto de la clase *Object* que permite saber la ubicación de dicho objeto dentro de la memoria, información suficiente para poder operar con él. Intuitivamente, la referencia es como una "flecha" que nos indica la posición del objeto que apunta:



## Listas enlazadas

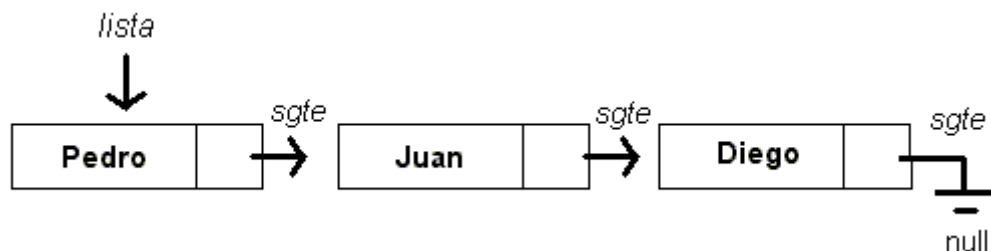
Una *lista enlazada* es una serie de *nodos*, conectados entre sí a través de una referencia, en donde se almacena la información de los elementos de la lista. Por lo tanto, los nodos de una lista enlazada se componen de dos partes principales:



```
class NodoLista
{
    Object elemento;
    NodoLista siguiente;
}
```

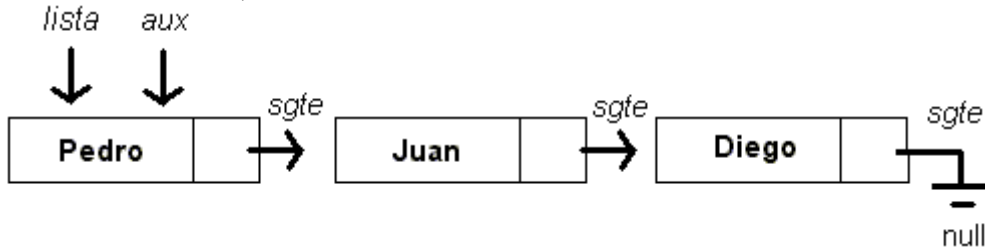
La referencia contenida en el nodo de una lista se denomina *siguiete*, pues indica en dónde se encuentra el siguiente elemento de la lista. El último elemento de la lista no tiene nodo siguiente, por lo que se dice que la referencia *siguiete* del último elemento es *null* (nula).

La siguiente figura muestra un ejemplo de una lista enlazada cuyos elementos son *strings*:

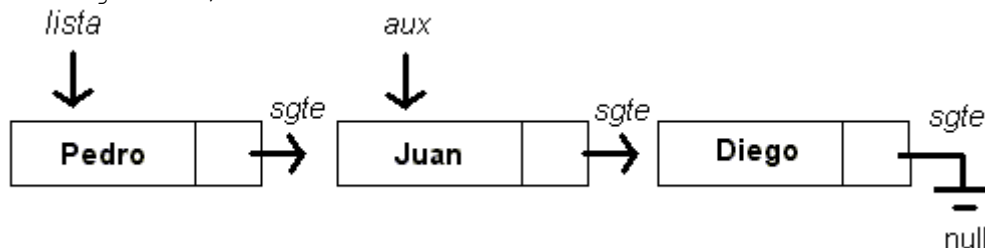


La referencia *lista* indica la posición del primer elemento de la lista y permite acceder a todos los elementos de ésta: basta con seguir las referencias al nodo siguiente para recorrer la lista.

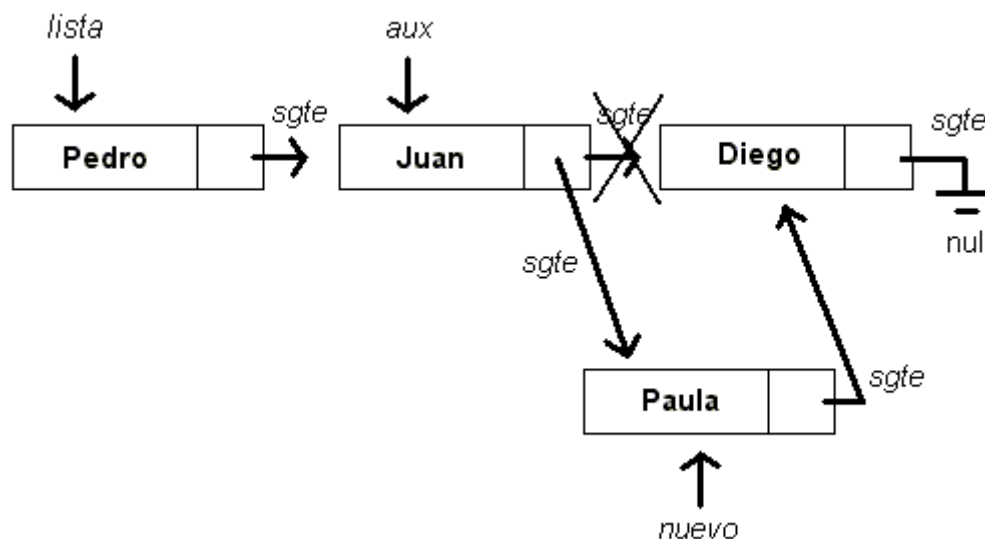
```
NodoLista aux=lista;
```



```
aux=aux.siguiente;
```



Siguiendo con el ejemplo anterior, para insertar un nuevo nodo justo delante del nodo referenciado por *aux* se deben modificar las referencias *siguiente* del nodo *aux* y del nodo a insertar.



```

NodoLista nuevo=new NodoLista(...);
// "nuevo" es la referencia del nodo a insertar en la lista
nuevo.siguiente=aux.siguiente;
aux.siguiente=nuevo;
// Notese que no es lo mismo realizar los cambios de referencia
// en un orden distinto al presentado, puesto que en ese caso
// se "pierde" la lista desde el nodo siguiente a aux
  
```

El procedimiento presentado a continuación es un ejemplo de cómo se programa el recorrido de una lista enlazada. Se supondrá que los objetos almacenados en cada nodo son *strings*:

```
void recorrido(NodoLista lista)
```



```

{
    NodoLista aux=lista;
    while (aux!=null)
    {
        System.out.println(aux.elemento);
        aux=aux.siguiente;
    }
}

```

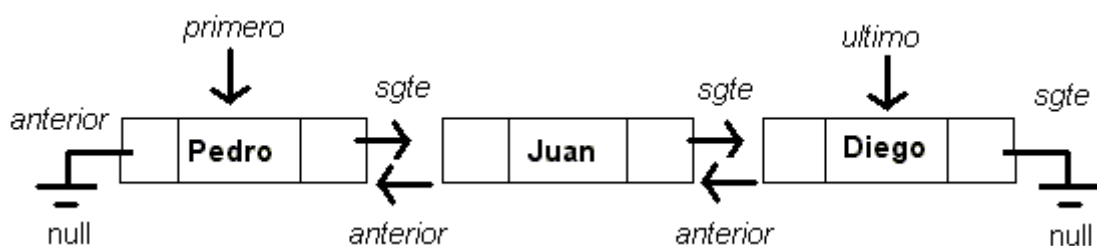
Para *invertir* el orden de la lista, es decir, que el último elemento de la lista ahora sea el primero, que el penúltimo elemento de la lista ahora sea el segundo, etc..., *modificando sólo las referencias y no el contenido de los nodos*, es necesario realizar una sola pasada por la lista, y en cada nodo visitado se modifica la referencia *siguiente* para que apunte al nodo anterior. Es necesario mantener referencias auxiliares para acordarse en donde se encuentra el nodo anterior y el resto de la lista que aún no ha sido modificada:

```

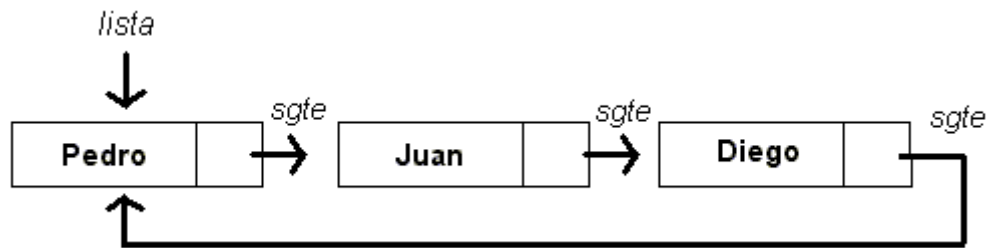
void invertir(NodoLista lista)
{
    NodoLista siguiente=lista;
    NodoLista anterior=null;
    while(lista!=null)
    {
        siguiente=lista.siguiente;
        lista.siguiente=anterior;
        anterior=lista;
        lista=siguiente;
    }
}

```

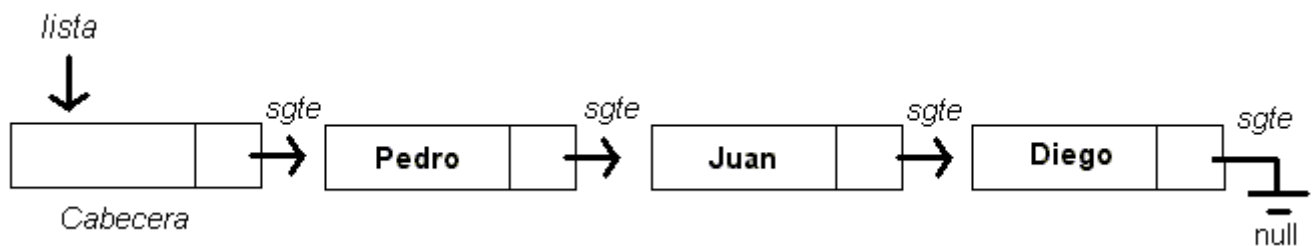
La implementación vista de los nodos también se conoce como *lista de enlace simple*, dado que sólo contiene una referencia al nodo siguiente y por lo tanto sólo puede recorrerse en un solo sentido. En una *lista de doble enlace* se agrega una segunda referencia al nodo previo, lo que permite recorrer la lista en ambos sentidos, y en general se implementa con una referencia al primer elemento y otra referencia al último elemento.



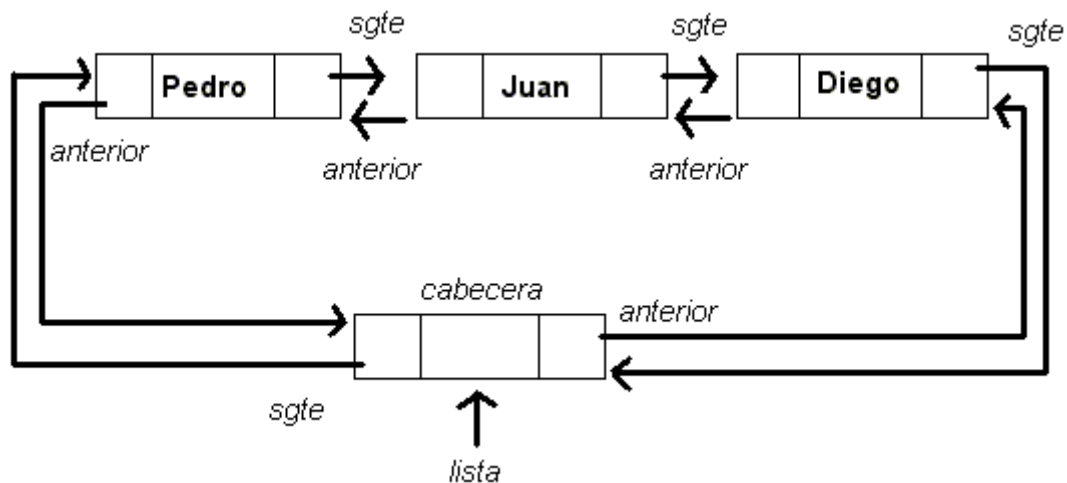
Una *lista circular* es aquella en donde la referencia siguiente del último nodo en vez de ser *null* apunta al primer nodo de la lista. El concepto se aplica tanto a listas de enlace simple como doblemente enlazadas.



En muchas aplicaciones que utilizan listas enlazadas es útil contar con un nodo *cabecera*, también conocido como *dummy* o *header*, que es un nodo "falso", ya que no contiene información relevante, y su referencia siguiente apunta al primer elemento de la lista. Al utilizar un nodo cabecera siempre es posible definir un nodo previo a cualquier nodo de la lista, definiendo que el previo al primer elemento es la cabecera.



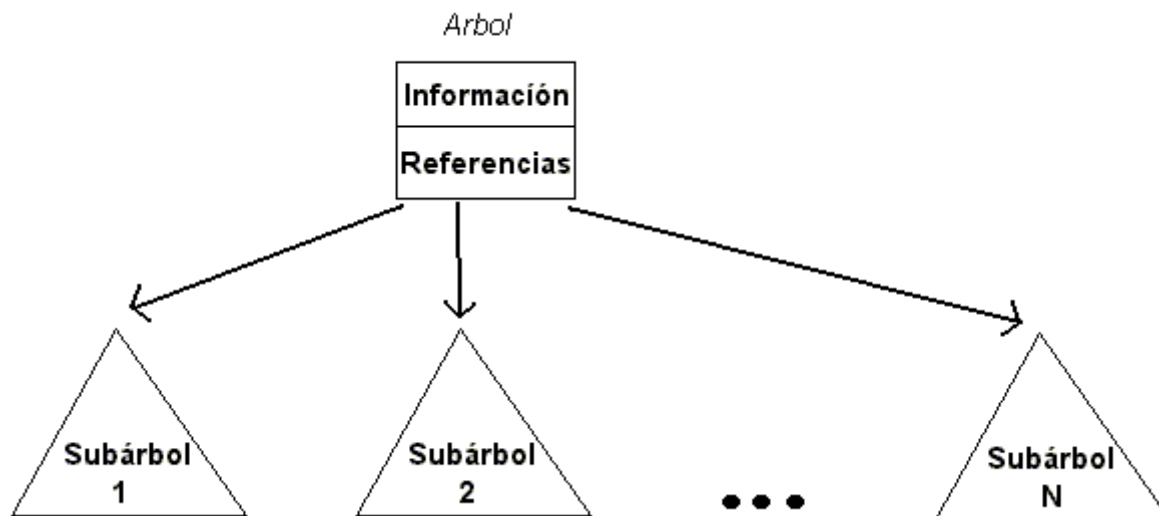
Si se utiliza un nodo cabecera en una lista de doble enlace ya no es necesario contar con las referencias *primero* y *último*, puesto que el nodo cabecera tiene ambas referencias: su referencia *siguiente* es el primer elemento de la lista, y su referencia *anterior* es el último elemento de la lista. De esta forma la lista de doble enlace queda circular de una manera natural.



## Árboles

Un *árbol* se define como una colección de nodos organizados en forma recursiva. Cuando hay 0 nodos se dice que el árbol está *vacío*, en caso contrario el árbol consiste en un nodo denominado *raíz*, el cual tiene 0 o más referencias a otros árboles, conocidos como *subárboles*. Las raíces de los subárboles se denominan *hijos* de la raíz, y

consecuentemente la raíz se denomina *padre* de las raíces de sus subárboles. Una visión gráfica de esta definición recursiva se muestra en la siguiente figura:

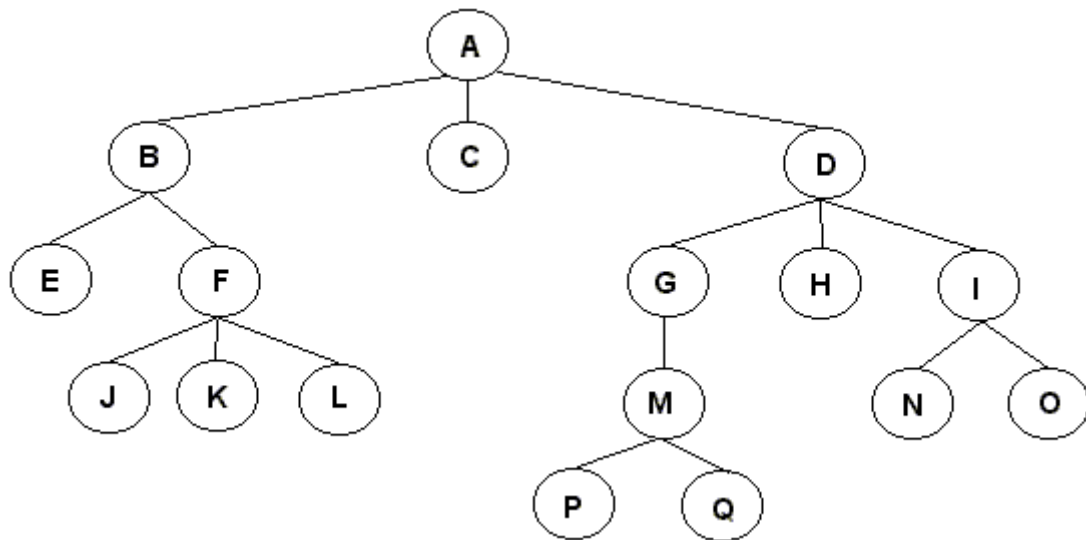


Los nodos que no poseen hijos se denominan *hojas*. Dos nodos que tienen el padre en común se denominan *hermanos*.

Un *camino* entre un nodo  $n_1$  y un nodo  $n_k$  está definido como la secuencia de nodos  $n_1, n_2, \dots, n_k$  tal que  $n_i$  es padre de  $n_{i+1}$ ,  $1 \leq i < k$ . El *largo del camino* es el número de referencias que componen el camino, que para el ejemplo son  $k-1$ . Existe un camino desde cada nodo del árbol a sí mismo y es de largo 0. *Nótese que en un árbol existe un único camino desde la raíz hasta cualquier otro nodo del árbol.* A partir del concepto de camino se definen los conceptos de *ancestro* y *descendiente*: un nodo  $n$  es *ancestro* de un nodo  $m$  si existe un camino desde  $n$  a  $m$ ; un nodo  $n$  es *descendiente* de un nodo  $m$  si existe un camino desde  $m$  a  $n$ .

Se define la *profundidad* del nodo  $n_k$  como el largo del camino entre la raíz del árbol y el nodo  $n_k$ . Esto implica que la profundidad de la raíz es siempre 0. La *altura* de un nodo  $n_k$  es el máximo largo de camino desde  $n_k$  hasta alguna hoja. Esto implica que la altura de toda hoja es 0. La *altura* de un árbol es igual a la altura de la raíz, y tiene el mismo valor que la profundidad de la hoja más profunda. La altura de un árbol vacío se define como -1.

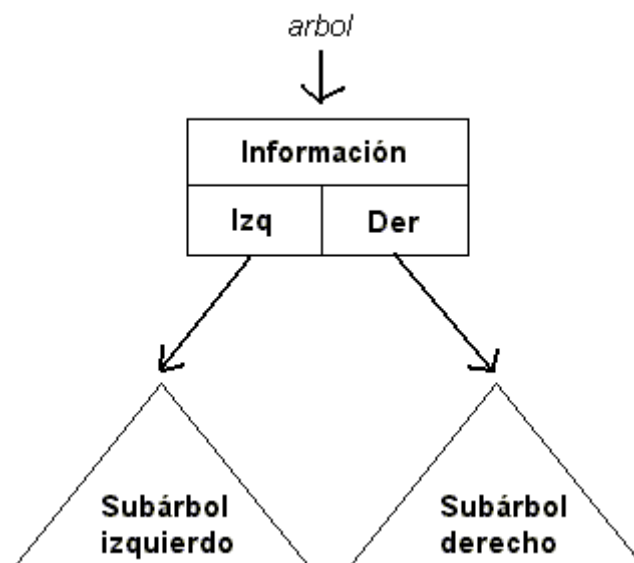
La siguiente figura muestra un ejemplo de los conceptos previamente descritos:



- A es la raíz del árbol.
- A es padre de B, C y D.
- E y F son hermanos, puesto que ambos son hijos de B.
- E, J, K, L, C, P, Q, H, N y O son las hojas del árbol.
- El camino desde A a J es único, lo conforman los nodos A-B-F-J y es de largo 3.
- D es ancestro de P, y por lo tanto P es descendiente de D.
- L no es descendiente de C, puesto que no existe un camino desde C a L.
- La profundidad de C es 1, de F es 2 y de Q es 4.
- La altura de C es 0, de F es 1 y de D es 3.
- La altura del árbol es 4 (largo del camino entre la raíz A y la hoja más profunda, P o Q).

## Árboles binarios

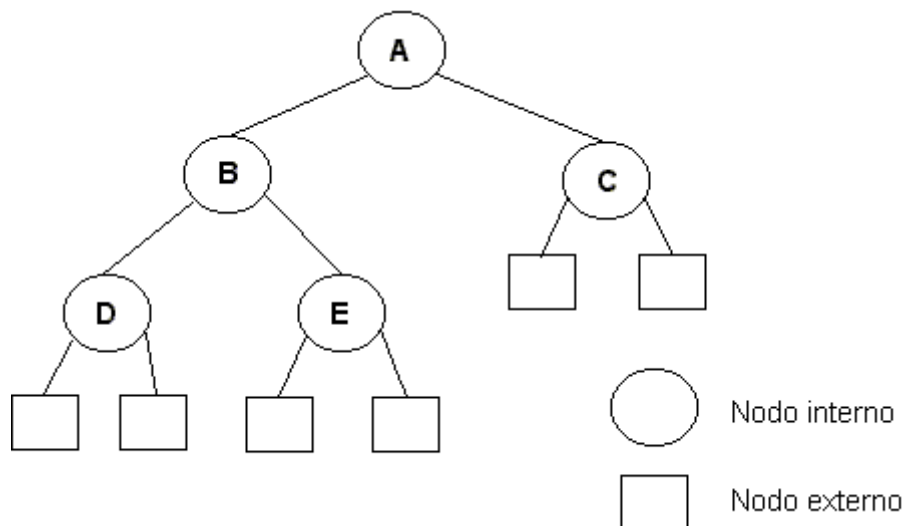
Un *árbol binario* es un árbol en donde cada nodo posee 2 referencias a subárboles (ni más, ni menos). En general, dichas referencias se denominan *izquierda* y *derecha*, y consecuentemente se define el *subárbol izquierdo* y *subárbol derecho* del árbol.



En este caso, la implementacion del nodo de un árbol binario es como sigue:

```
class NodoArbolBinario
{
    Object elemento;
    NodoArbolBinario izq;
    NodoArbolBinario der;
}
```

Los nodos en sí que conforman un árbol binario se denominan *nodos internos*, y todas las referencias que son *null* se denominan *nodos externos*.



## Propiedades de los árboles binarios

Propiedad 1:

Si se define  $i$  = número de nodos internos,  $e$  = número de nodos externos, entonces se tiene que:

$$e = i + 1$$

Demostración: inducción sobre  $i$  (ejercicio).

Propiedad 2:

Sea  $n$  = número de nodos internos. Se define:

- $I_n$  = suma del largo de los caminos desde la raíz a cada nodo interno (largo de caminos internos).
- $E_n$  = suma del largo de los caminos desde la raíz a cada nodo externo (largo de caminos externos).

Se tiene que:

$$E_n = I_n + 2n$$

Demostración: inducción sobre  $n$  (ejercicio).

Propiedad 3:

¿Cuántos árboles binarios distintos se pueden construir con  $n$  nodos internos?

$n$	$b_n$
0	1
1	1
2	2
3	5

¿ $b_n$ ?

$$b_0 = 1$$

$$b_{n+1} = \sum_{i=0}^n b_i \cdot b_{n-i}$$

Por ejemplo:  $b_4 = b_0*b_3 + b_1*b_2 + b_2*b_1 + b_3*b_0 = 5 + 2 + 2 + 5 = 14$ .

Este tipo de ecuaciones se puede resolver y la solución es la siguiente:

$$b_n = \frac{1}{n+1} \binom{2n}{n}$$

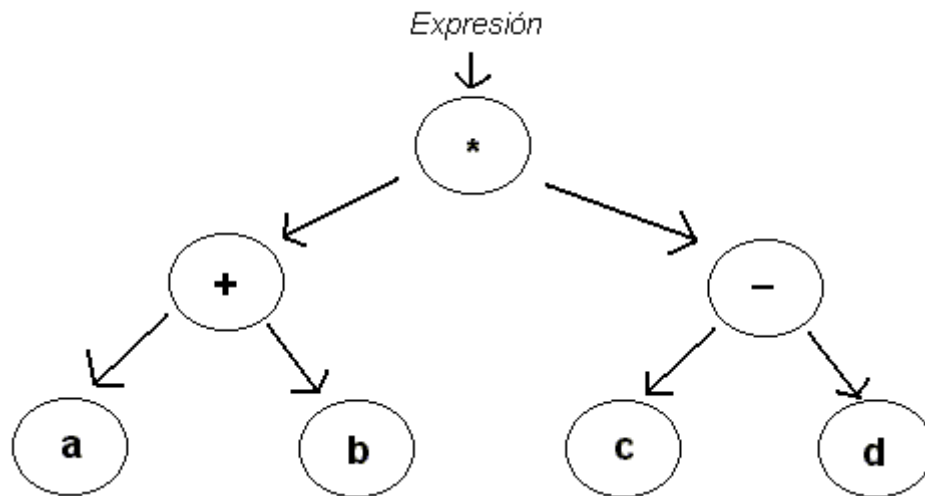
La serie de numeros que genera  $b_n$  se conoce como *números de Catalan*.  
Asintóticamente:

$$b_n \approx \frac{4^n}{n\sqrt{\pi n}}$$

### Ejemplo: árboles de expresiones matemáticas

La siguiente figura muestra un ejemplo de un *árbol de expresiones matemáticas*. En un árbol de expresiones las hojas corresponden a los *operandos* de la expresión (variables o constantes), mientras que los nodos restantes contienen *operadores*. Dado que los

operadores matemáticos son binarios (o unarios como en el caso del operador signo -), un árbol de expresiones resulta ser un árbol binario.



Un árbol de expresiones se puede evaluar de la siguiente forma:

- Si la raíz del árbol es una constante o una variable se retorna el valor de ésta.
- Si la raíz resulta ser un operador, entonces recursivamente se evalúan los subárboles izquierdo y derecho, y se retorna el valor que resulta al operar los valores obtenidos de las evaluaciones de los subárboles con el operador respectivo.

### Recorridos de árboles binarios

Existen tres formas principales para recorrer un árbol binario en forma recursiva. Estas son:

- *Preorden*: raíz - subárbol izquierdo - subárbol derecho.
- *Inorden*: subárbol izquierdo - raíz - subárbol derecho.
- *Postorden*: subárbol izquierdo - subárbol derecho - raíz.

Por ejemplo, al recorrer el árbol de expresiones anterior en *preorden* se obtiene:

$$* + a b - c d$$

Al recorrer el árbol en *inorden* se obtiene:

$$a + b * c - d$$

Al recorrer el árbol en *postorden* se obtiene:

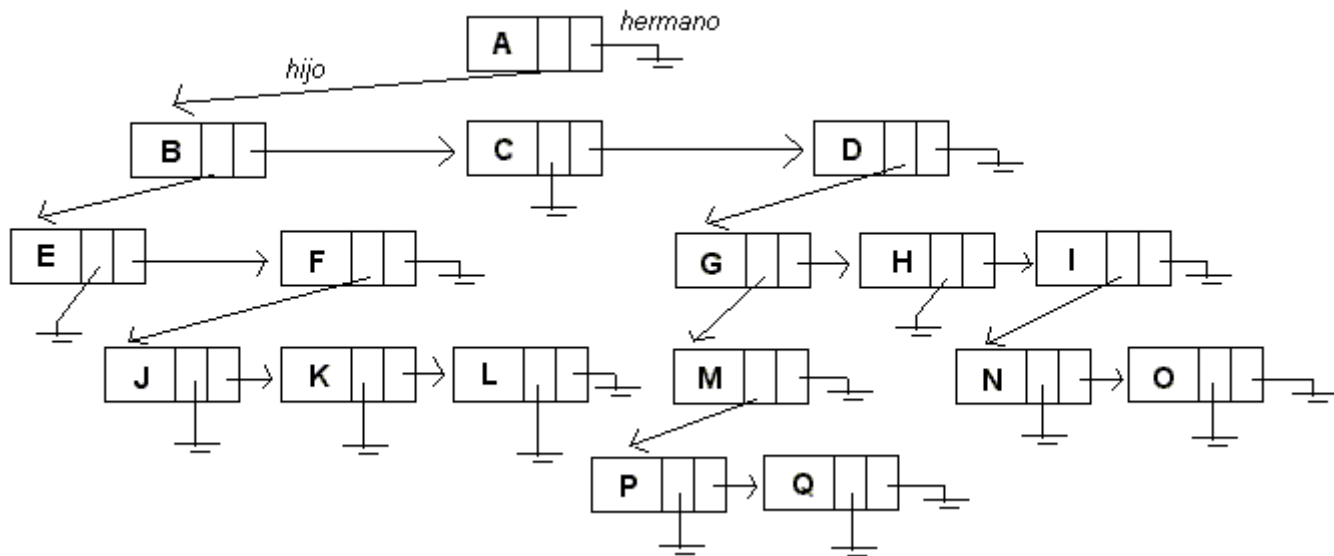
$$a b + c d - *$$

La expresión que se obtiene con el recorrido en *postorden* se conoce como *notación polaca inversa*.

## Árboles generales

En un árbol general cada nodo puede poseer un número indeterminado de hijos. La implementación de los nodos en este caso se realiza de la siguiente manera: como no se sabe de antemano cuantos hijos tiene un nodo en particular se utilizan dos referencias, una a su primer hijo y otra a su hermano más cercano. La raíz del árbol necesariamente tiene la referencia a su hermano como *null*.

```
class NodoArbolGeneral
{
    Object elemento;
    NodoArbolGeneral hijo;
    NodoArbolGeneral hermano;
}
```



Nótese que todo árbol general puede representarse como un árbol binario, con la salvedad que el hijo derecho de la raíz es siempre *null*. Si se permite que la raíz del árbol tenga hermanos, lo que se conoce como *bosque*, entonces se tiene que el conjunto de los bosques generales es isomorfo al conjunto de los árboles binarios. En efecto, las propiedades vistas en los árboles binarios se siguen cumpliendo en los árboles generales.



# Tipos de datos abstractos

1. [TDA lista.](#)
2. [TDA pila.](#)
3. [TDA cola.](#)
4. [TDA cola de prioridad](#)

Un *Tipo de dato abstracto* (en adelante *TDA*) es un conjunto de datos u objetos al cual se le asocian *operaciones*. El TDA provee de una interfaz con la cual es posible realizar las operaciones permitidas, abstrayéndose de la manera en como estén implementadas dichas operaciones. Esto quiere decir que un mismo TDA puede ser implementado utilizando distintas estructuras de datos y proveer la misma funcionalidad.

El paradigma de orientación a objetos permite el *encapsulamiento* de los datos y las operaciones mediante la definición de *clases* e *interfaces*, lo cual permite *ocultar* la manera en cómo ha sido implementado el TDA y solo permite el acceso a los datos a través de las operaciones provistas por la interfaz.

En este capítulo se estudiarán TDA básicos como lo son las *listas*, *pilas* y *colas*, y se mostrarán algunos usos prácticos de estos TDA.

## TDA lista

Una *lista* se define como una serie de  $N$  elementos  $E_1, E_2, \dots, E_N$ , ordenados de manera consecutiva, es decir, el elemento  $E_k$  (que se denomina *elemento k-ésimo*) es previo al elemento  $E_{k+1}$ . Si la lista contiene 0 elementos se denomina como *lista vacía*.

Las operaciones que se pueden realizar en la lista son: insertar un elemento en la posición  $k$ , borrar el  $k$ -ésimo elemento, buscar un elemento dentro de la lista y preguntar si la lista esta vacía.

Una manera simple de implementar una lista es utilizando un arreglo. Sin embargo, las operaciones de inserción y borrado de elementos en arreglos son ineficientes, puesto que para insertar un elemento en la parte media del arreglo es necesario mover todos los elementos que se encuentren delante de él, para hacer espacio, y al borrar un elemento es necesario mover todos los elementos para ocupar el espacio desocupado. Una implementación más eficiente del TDA se logra utilizando listas enlazadas.

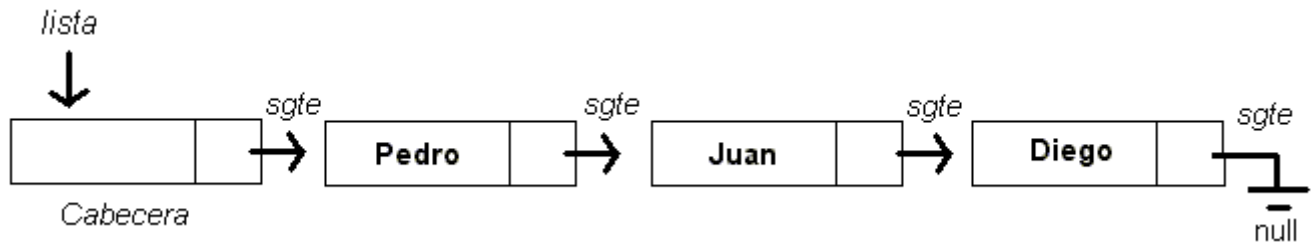
A continuación se presenta una implementación en Java del TDA utilizando listas enlazadas y sus operaciones asociadas:

- **estaVacía()**: devuelve *verdadero* si la lista esta vacía, falso en caso contrario.
- **insertar(x, k)**: inserta el elemento  $x$  en la  $k$ -ésima posición de la lista.
- **buscar(x)**: devuelve la posición en la lista del elemento  $x$ .
- **buscarK(k)**: devuelve el  $k$ -ésimo elemento de la lista.
- **eliminar(x)**: elimina de la lista el elemento  $x$ .

En la implementación con listas enlazadas es necesario tener en cuenta algunos detalles importantes: si solamente se dispone de la referencia al primer elemento, el añadir o

remove en la primera posición es un caso especial, puesto que la referencia a la lista enlazada debe modificarse según la operación realizada. Además, para eliminar un elemento en particular es necesario conocer el elemento que lo antecede, y en este caso, ¿qué pasa con el primer elemento, que no tiene un predecesor?

Para solucionar estos inconvenientes se utiliza la implementación de lista enlazada con nodo cabecera. Con esto, todos los elementos de la lista tendrán un elemento previo, puesto que el previo del primer elemento es la cabecera. Una lista vacía corresponde, en este caso, a una cabecera cuya referencia siguiente es *null*.



Los archivos [NodoLista.java](#), [IteradorLista.java](#) y [Lista.java](#) contienen una implementación completa del TDA lista. La clase *NodoLista* implementa los nodos de la lista enlazada, la clase *Lista* implementa las operaciones de la lista propiamente tal, y la clase *IteradorLista* implementa objetos que permiten recorrer la lista y posee la siguiente interfaz:

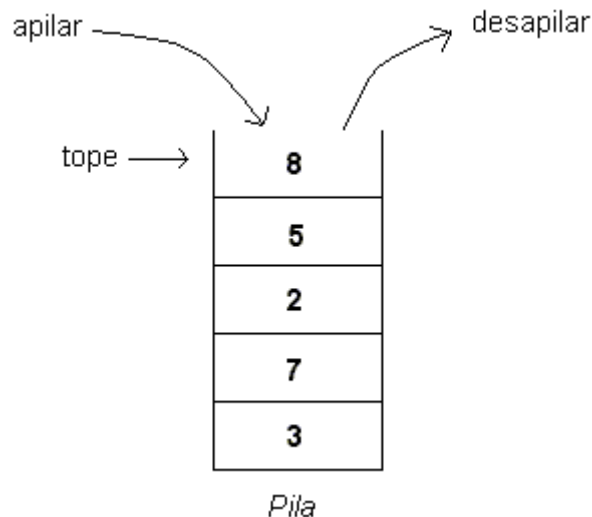
- **avanzar()**: avanza el iterador al siguiente nodo de la lista.
- **obtener()**: retorna el elemento del nodo en donde se encuentra el iterador.

Costo de las operaciones en tiempo:

- Insertar/eliminar elemento en k-ésima posición:  $O(k)$  (¿Se puede hacer en  $O(1)$ ?).
- Buscar elemento  $x$ :  $O(N)$  (promedio).

## TDA pila

Una *pila* (*stack* o *pushdown* en inglés) es una lista de elementos de la cual sólo se puede extraer el último elemento insertado. La posición en donde se encuentra dicho elemento se denomina *tope* de la pila. También se conoce a las pilas como *listas LIFO* (LAST IN - FIRST OUT: el último que entra es el primero que sale).



La interfaz de este TDA provee las siguientes operaciones:

- **apilar(x)**: inserta el elemento  $x$  en el tope de la pila (**push** en inglés).
- **desapilar()**: retorna el elemento que se encuentre en el tope de la pila y lo elimina de ésta (**pop** en inglés).
- **tope()**: retorna el elemento que se encuentre en el tope de la pila, pero sin eliminarlo de ésta (**top** en inglés).
- **estaVacia()**: retorna *verdadero* si la pila no contiene elementos, *falso* en caso contrario (**isEmpty** en inglés).

*Nota*: algunos autores definen **desapilar** como sacar el elemento del tope de la pila sin retornarlo.

## Implementación del TDA pila

A continuación se muestran dos maneras de implementar una pila: utilizando un arreglo y utilizando una lista enlazada. En ambos casos el costo de las operaciones es de  $O(1)$ .

### Implementación utilizando arreglos

Para implementar una pila utilizando un arreglo, basta con definir el arreglo del tipo de dato que se almacenará en la pila. Una variable de instancia indicará la posición del tope de la pila, lo cual permitirá realizar las operaciones de inserción y borrado, y también permitirá saber si la pila está vacía, definiendo que dicha variable vale **-1** cuando no hay elementos en el arreglo.

```
class PilaArreglo
{
    private Object[] arreglo;
    private int tope;
    private int MAX_ELEM=100; // maximo numero de elementos en la pila

    public PilaArreglo()
    {
        arreglo=new Object[MAX_ELEM];
        tope=-1; // inicialmente la pila esta vacía
    }
}
```

```

public void apilar(Object x)
{
    if (tope+1<MAX_ELEM) // si esta llena se produce OVERFLOW
    {
        tope++;
        arreglo[tope]=x;
    }
}

public Object desapilar()
{
    if (!estaVacia()) // si esta vacia se produce UNDERFLOW
    {
        Object x=arreglo[tope];
        tope--;
        return x;
    }
}

public Object tope()
{
    if (!estaVacia()) // si esta vacia es un error
    {
        Object x=arreglo[tope];
        return x;
    }
}

public boolean estaVacia()
{
    if (tope==-1)
    {
        return true;
    }
    else
    {
        return false;
    }
}
}

```

El inconveniente de esta implementación es que es necesario fijar de antemano el número máximo de elementos que puede contener la pila, *MAX\_ELEM*, y por lo tanto al apilar un elemento es necesario controlar que no se inserte un elemento si la pila esta llena. Sin embargo, en Java es posible solucionar este problema creando un nuevo arreglo más grande que el anterior, el doble por ejemplo, y copiando los elementos de un arreglo a otro:

```

public void apilar(Object x)
{
    if (tope+1<MAX_ELEM) // si esta llena se produce OVERFLOW
    {
        tope++;
        arreglo[tope]=x;
    }
    else
    {
        MAX_ELEM=MAX_ELEM*2;
        Object[] nuevo_arreglo=new Object[MAX_ELEM];
    }
}

```

```

        for (int i=0; i<arreglo.length; i++)
        {
            nuevo_arreglo[i]=arreglo[i];
        }
        tope++;
        nuevo_arreglo[tope]=x;
        arreglo=nuevo_arreglo;
    }
}

```

## Implementación utilizando listas enlazadas

En este caso no existe el problema de tener que fijar el tamaño máximo de la pila (pero está acotado por la cantidad de memoria disponible). La implementación es bastante simple: los elementos se insertan al principio de la lista (apilar) y siempre se extrae el primer elemento de la lista (desapilar y tope), por lo que basta con tener una referencia al principio de la lista enlazada. Si dicha referencia es *null*, entonces la pila está vacía.

```

class PilaLista
{
    private NodoLista lista;

    public PilaLista()
    {
        lista=null;
    }

    public void apilar(Object x)
    {
        lista=new NodoLista(x, lista);
    }

    public Object desapilar() // si esta vacia se produce UNDERFLOW
    {
        if (!estaVacia())
        {
            Object x=lista.elemento;
            lista=lista.siguiente;
            return x;
        }
    }

    public Object tope()
    {
        if (!estaVacia()) // si esta vacia es un error
        {
            Object x=lista.elemento;
            return x;
        }
    }

    public boolean estaVacia()
    {
        return lista==null;
    }
}

```

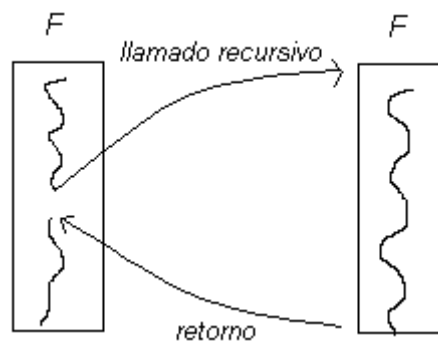
Dependiendo de la aplicación que se le de a la pila es necesario definir que acción realizar en caso de que ocurra *overflow* (rebalse de la pila) o *underflow* (intentar

desapilar cuando la pila esta vacía). Java posee un mecanismo denominado *excepciones*, que permite realizar acciones cuando se producen ciertos eventos específicos (como por ejemplo *overflow* o *underflow* en una pila).

En ambas implementaciones el costo de las operaciones que provee el TDA tienen costo  $O(1)$ .

## Ejemplo de uso: eliminación de recursividad

Suponga que una función  $F$  realiza un llamado recursivo dentro de su código, lo que se ilustra en la siguiente figura:



Si la llamada recursiva es lo último que hace la función  $F$ , entonces dicha llamada se puede substituir por un ciclo *while*. Este caso es conocido como *tail recursion* y en lo posible hay que evitarlo en la programación, ya que cada llamada recursiva ocupa espacio en la memoria del computador, y en el caso del *tail recursion* es muy simple eliminarla. Por ejemplo:

```
void imprimir(int[] a, int j) // versión recursiva
{
    if (j<a.length)
    {
        System.out.println(a[j]);
        imprimir(a, j+1); // tail recursion
    }
}

void imprimir(int[] a, int j) // versión iterativa
{
    while (j<a.length)
    {
        System.out.println(a[j]);
        j=j+1;
    }
}
```

En el caso general, cuando el llamado recursivo se realiza en medio de la función  $F$ , la recursión se puede eliminar utilizando una pila.

Por ejemplo: recorrido en preorden de un árbol binario.

```
// "raiz" es la referencia a la raiz del árbol
// llamado inicial: preorden(raiz)
```

```

// version recursiva

void preorden(Nodo nodo)
{
    if (nodo!=null)
    {
        System.out.print(nodo.elemento);
        preorden(nodo.izq);
        preorden(nodo.der);
    }
}

// primera version iterativa

void preorden(Nodo nodo)
{
    Nodo aux;
    Pila pila=new Pila(); // pila de nodos
    pila.apilar(nodo);
    while(!pila.estaVacia()) // mientras la pila no este vacia
    {
        aux=pila.desapilar();
        if (aux!=null)
        {
            System.out.print(aux.elemento);
            // primero se apila el nodo derecho y luego el izquierdo
            // para mantener el orden correcto del recorrido
            // al desapilar los nodos
            pila.apilar(aux.der);
            pila.apilar(aux.izq);
        }
    }
}

// segunda version iterativa
// dado que siempre el ultimo nodo apilado dentro del bloque if es
// aux.izq podemos asignarlo directamente a aux hasta que éste sea
// null, es decir, el bloque if se convierte en un bloque while
// y se cambia el segundo apilar por una asignacion de la referencia

void preorden(Nodo nodo)
{
    Nodo aux;
    Pila pila=new Pila(); // pila de nodos
    pila.apilar(nodo);
    while(!pila.estaVacia()) // mientras la pila no este vacia
    {
        aux=pila.desapilar();
        while (aux!=null)
        {
            System.out.print(aux.elemento);
            pila.apilar(aux.der);
            aux=aux.izq;
        }
    }
}

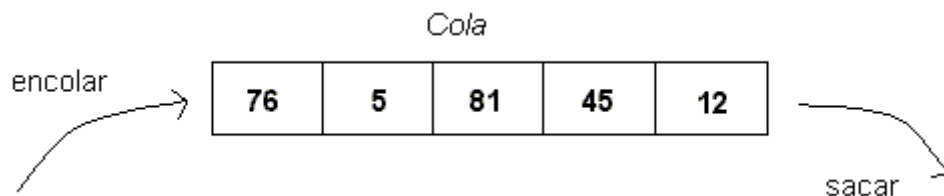
```

Si bien los programas no recursivos son más eficientes que los recursivos, la eliminación de recursividad (excepto en el caso de *tail recursion*) le quita claridad al código del programa. Por lo tanto:

- A menudo es conveniente eliminar el *tail recursion*.
- Un método recursivo es menos eficiente que uno no recursivo, pero sólo en pocas oportunidades vale la pena eliminar la recursión.

## TDA cola

Una *cola* (*queue* en inglés) es una lista de elementos en donde siempre se insertan nuevos elementos al final de la lista y se extraen elementos desde el inicio de la lista. También se conoce a las colas como *listas FIFO* (FIRST IN - FIRST OUT: el primero que entra es el primero que sale).



Las operaciones básicas en una cola son:

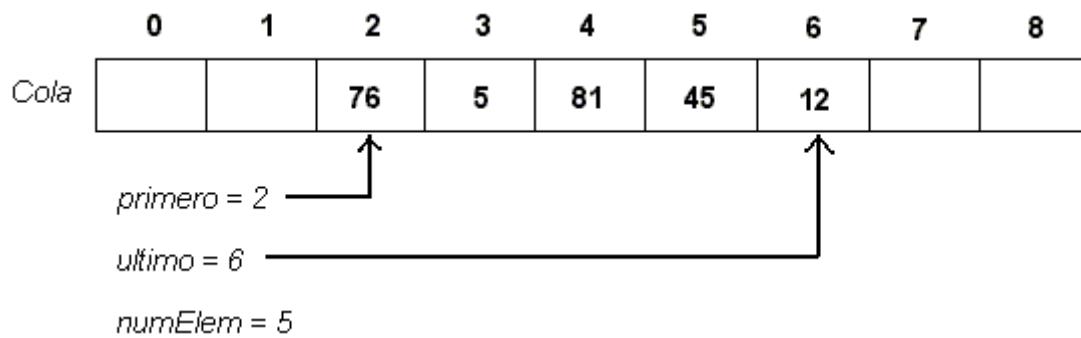
- **encolar(x)**: inserta el elemento  $x$  al final de la cola (**enqueue** en inglés).
- **sacar()**: retorna el elemento que se ubica al inicio de la cola (**dequeue** en inglés).
- **estaVacía()**: retorna *verdadero* si la cola esta vacía, *falso* en caso contrario.

Al igual que con el TDA pila, una cola se puede implementar tanto con arreglos como con listas enlazadas. A continuación se verá la implementación usando un arreglo.

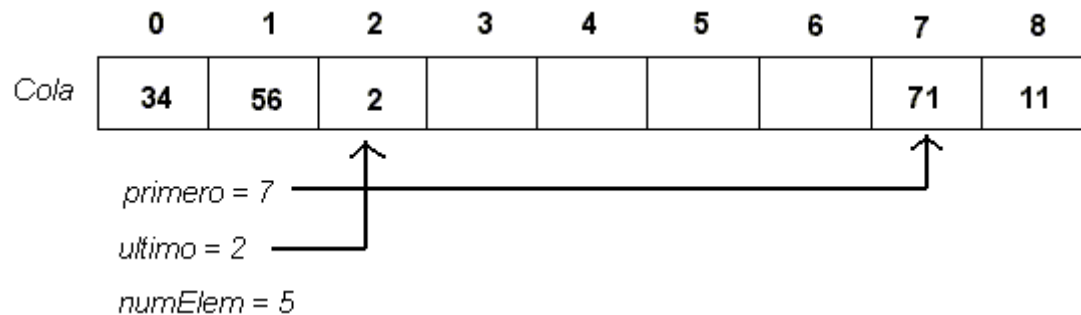
Las variables de instancia necesarias en la implementación son:

- *primero*: indica el índice de la posición del primer elemento de la cola, es decir, la posición del elemento a retornar cuando se invoque **sacar**.
- *ultimo*: indica el índice de la posición de último elemento de la cola. Si se invoca **encolar**, el elemento debe ser insertado en el casillero siguiente al que indica la variable.
- *numElem*: indica cuántos elementos posee la cola. Definiendo *MAX\_ELEM* como el tamaño máximo del arreglo, y por lo tanto de la cola, entonces la cola esta vacía si *numElem*==0 y está llena si *numElem*==*MAX\_ELEM*.





Un detalle faltante es el siguiente: ¿qué pasa si la variable *ultimo* sobrepasa el rango de índices del arreglo? Esto se soluciona definiendo que si después de insertar un elemento el índice  $\text{ultimo} == \text{MAX\_ELEM}$ , entonces se asigna  $\text{ultimo} = 0$ , y los siguientes elementos serán insertados al comienzo del arreglo. Esto no produce ningún efecto en la lógica de las operaciones del TDA, pues siempre se saca el elemento referenciado por el índice *primero*, aunque en valor absoluto  $\text{primero} > \text{ultimo}$ . Este enfoque es conocido como *implementación con arreglo circular*, y la forma más fácil de implementarlo es haciendo la aritmética de subíndices módulo *MAX\_ELEM*.



```

class ColaArreglo
{
    private Object[] arreglo;
    private int primero, ultimo, numElem;
    private int MAX_ELEM=100; // maximo numero de elementos en la cola

    public ColaArreglo()
    {
        arreglo=new Object[MAX_ELEM];
        primero=0;
        ultimo=-1;
        numElem=0;
    }

    public void encolar(Object x)
    {
        if (numElem<=MAX_ELEM) // si esta llena se produce OVERFLOW
        {
            ultimo=(ultimo+1)%MAX_ELEM;
            arreglo[ultimo]=x;
            numElem++;
        }
    }

    public Object sacar()
    {
        if (!estaVacia()) // si esta vacia se produce UNDERFLOW
        {

```

```

        Object x=arreglo[primero];
        primero=(primero+1)%MAX_ELEM;
        numElem--;
        return x;
    }
}

public boolean estaVacía()
{
    return num_elem==0;
}
}

```

Nuevamente en este caso, dependiendo de la aplicación, se debe definir qué hacer en caso de producirse OVERFLOW o UNDERFLOW.

Con esta implementación, todas las operaciones del TDA cola tienen costo  $O(1)$ .

## TDA Cola de Prioridad

Una *cola de prioridad* es un tipo de datos abstracto que almacena un conjunto de datos que poseen una llave perteneciente a algún conjunto ordenado, y permite *insertar* nuevos elementos y *extraer el máximo* (o el mínimo, en caso de que la estructura se organice con un criterio de orden inverso).

Es frecuente interpretar los valores de las llaves como prioridades, con lo cual la estructura permite insertar elementos de prioridad cualquiera, y extraer el de mejor prioridad.

Dos formas simples de implementar colas de prioridad son:

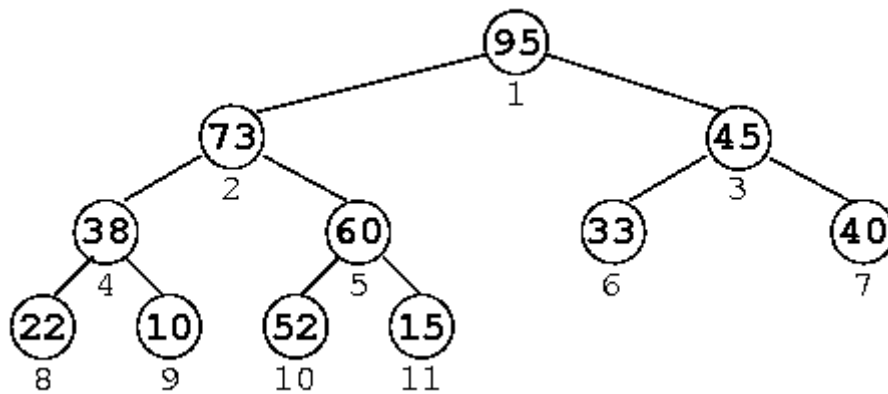
- Una lista ordenada:
  - Inserción:  $O(n)$
  - Extracción de máximo:  $O(1)$
- Una lista desordenada:
  - Inserción:  $O(1)$
  - Extracción de máximo:  $O(n)$

## Heaps

Un heap es un árbol binario de una forma especial, que permite su almacenamiento en un arreglo sin usar punteros.

Un heap tiene todos sus niveles llenos, excepto posiblemente el de más abajo, y en este último los nodos están lo más a la izquierda posible.

Ejemplo:



La numeración por niveles (indicada bajo cada nodo) son los subíndices en donde cada elemento sería almacenado en el arreglo. En el caso del ejemplo, el arreglo sería:

<b>95</b>	<b>73</b>	<b>45</b>	<b>38</b>	<b>60</b>	<b>33</b>	<b>40</b>	<b>22</b>	<b>10</b>	<b>52</b>	<b>15</b>
1	2	3	4	5	6	7	8	9	10	11

La característica que permite que un heap se pueda almacenar sin punteros es que, si se utiliza la numeración por niveles indicada, entonces la relación entre padres e hijos es:

Hijos del nodo  $j = \{2*j, 2*j+1\}$

Padre del nodo  $k = \text{floor}(k/2)$

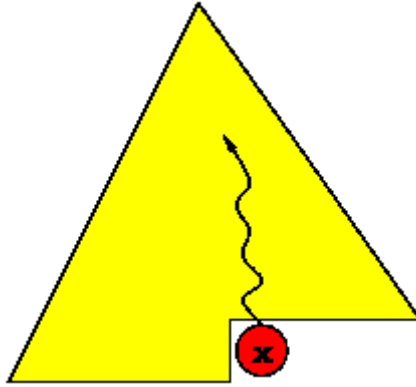
Un heap puede utilizarse para implementar una cola de prioridad almacenando los datos de modo que las llaves estén siempre ordenadas de arriba a abajo (a diferencia de un árbol de búsqueda binaria, que ordena sus llaves de izquierda a derecha). En otras palabras, el padre debe tener siempre mayor prioridad que sus hijos (ver ejemplo).

### Implementación de las operaciones básicas

#### *Inserción:*

La inserción se realiza agregando el nuevo elemento en la primera posición libre del heap, esto es, el próximo nodo que debería aparecer en el recorrido por niveles o, equivalentemente, un casillero que se agrega al final del arreglo.

Después de agregar este elemento, la *forma* del heap se preserva, pero la restricción de orden no tiene por qué cumplirse. Para resolver este problema, si el nuevo elemento es mayor que su padre, se intercambia con él, y ese proceso se repite mientras sea necesario. Una forma de describir esto es diciendo que el nuevo elemento "trepa" en el árbol hasta alcanzar el nivel correcto según su prioridad.



El siguiente trozo de programa muestra el proceso de inserción de un nuevo elemento  $x$ :

```

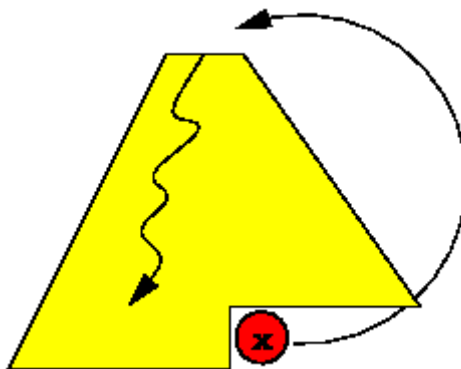
a[++n]=x;
for(j=n; j>1 && a[j]>a[j/2]; j/=2)
{ # intercambiamos con el padre
  t=a[j];
  a[j]=a[j/2];
  a[j/2]=t;
}

```

El proceso de inserción, en el peor caso, toma un tiempo proporcional a la altura del árbol, esto es,  $O(\log n)$ .

### *Extracción del máximo*

El máximo evidentemente está en la raíz del árbol (casillero 1 del arreglo). Al sacarlo de ahí, podemos imaginar que ese lugar queda vacante. Para llenarlo, tomamos al *último* elemento del heap y lo trasladamos al lugar vacante. En caso de que no esté bien ahí de acuerdo a su prioridad (¡que es lo más probable!), lo hacemos descender intercambiándolo siempre con el mayor de sus hijos. Decimos que este elemento "se hunde" hasta su nivel de prioridad.



El siguiente trozo de programa implementa este algoritmo:

```

m=a[1]; # La variable m lleva el máximo
a[1]=a[n--]; # Movemos el último a la raíz y achicamos el heap
j=1;

```

```
while(2*j<=n) # mientras tenga algún hijo
{
    k=2*j; # el hijo izquierdo
    if(k+1<=n && a[k+1]>a[k])
        k=k+1; # el hijo derecho es el mayor
    if(a[j]>a[k])
        break; # es mayor que ambos hijos
    t=a[j];
    a[j]=a[k];
    a[k]=t;
    j=k; # lo intercambiamos con el mayor hijo
}
```

Este algoritmo también demora un tiempo proporcional a la altura del árbol en el peor caso, esto es,  $O(\log n)$ .

# TDA diccionario

1. [Implementaciones sencillas.](#)
  - [Búsqueda binaria.](#)
  - [Búsqueda secuencial con probabilidades de acceso no uniforme.](#)
2. [Arboles de búsqueda binaria.](#)
3. [Arboles AVL.](#)
4. [Arboles 2-3.](#)
5. [Arboles B.](#)
6. [Arboles digitales.](#)
7. [Arboles de búsqueda digital.](#)
8. [Skip lists.](#)
9. [ABB óptimos.](#)
10. [Splay Trees.](#)
11. [Hashing.](#)
  - [Encadenamiento.](#)
  - [Direccionamiento abierto.](#)
  - [Hashing en memoria secundaria.](#)

Dado un conjunto de elementos  $\{X_1, X_2, \dots, X_N\}$ , todos distintos entre sí, se desea almacenarlos en una estructura de datos que permita la implementación eficiente de las operaciones:

- **búsqueda(X):** dado un elemento  $X$ , conocido como *llave de búsqueda*, encontrarlo dentro del conjunto o decir que no está.
- **inserción(X):** agregar un nuevo elemento  $X$  al conjunto.
- **eliminación(X):** eliminar el elemento  $X$  del conjunto.

Estas operaciones describen al TDA *diccionario*. En el presente capítulo se verán distintas implementaciones de este TDA y se estudiarán las consideraciones de eficiencia de cada una de dichas implementaciones.

## Implementaciones sencillas

Una manera simple de implementar el TDA diccionario es utilizando una *lista*, la cual permite implementar la *inserción* de nuevos elementos de manera muy eficiente, definiendo que siempre se realiza al comienzo de la lista. El problema que tiene esta implementación es que las operaciones de *búsqueda* y *eliminación* son ineficientes, puesto que como en la lista los elementos están desordenados es necesario realizar una *búsqueda secuencial*. Por lo tanto, los costos asociados a cada operación son:

- búsqueda:  $O(n)$  (búsqueda secuencial).
- inserción:  $O(1)$  (insertando siempre al comienzo de la lista).
- eliminación:  $O(n)$  (búsqueda +  $O(1)$ ).

Otra manera sencilla de implementar el TDA es utilizando un *arreglo ordenado*. En este caso, la operación de *inserción* y *eliminación* son ineficientes, puesto que para mantener el orden dentro del arreglo es necesario "correr" los elementos para dejar espacio al

insertar un nuevo elemento. Sin embargo, la ventaja que tiene mantener el orden es que es posible realizar una *búsqueda binaria* para encontrar el elemento buscado.

## Búsqueda binaria

Suponga que se dispone del arreglo  $a$ , de tamaño  $n$ , en donde se tiene almacenado el conjunto de elementos ordenados de menor a mayor. Para buscar un elemento  $x$  dentro del arreglo se debe:

- Buscar el índice de la posición media del arreglo en donde se busca el elemento, que se denotará  $m$ . Inicialmente,  $m = n/2$ .
- Si  $a[m] = x$  se encontró el elemento (fin de la búsqueda), en caso contrario se sigue buscando en el lado derecho o izquierdo del arreglo dependiendo si  $a[m] < x$  o  $a[m] > x$  respectivamente.

*Costo de la búsqueda binaria:*

$$T(n) = 1 + T(n/2) \text{ (aproximadamente)}$$

$$T(n) = 2 + T(n/4)$$

$$T(n) = 3 + T(n/8)$$

...

$$T(n) = k + T(n/2^k) \text{ para todo } k \geq 0$$

Escogiendo  $k = \log_2 n$ :

$$\Rightarrow T(n) = \log_2 n + T(1) = 1 + \log_2 n = O(\log n).$$

## Programación de la búsqueda binaria

La variable  $i$  representa el primer casillero del arreglo en donde es posible que se encuentre el elemento  $x$ , la variable  $j$  representa el último casillero del arreglo hasta donde  $x$  puede pertenecer, con  $j \geq i$ .

Inicialmente:  $i = 0$  y  $j = n-1$ .



En cada iteración:

- Si el conjunto es vacío ( $j-i < 0$ ), o sea si  $j < i$ , entonces el elemento  $x$  no está en el conjunto (búsqueda infructuosa).
- En caso contrario,  $m = (i+j)/2$ . Si  $x = a[m]$ , el elemento fue encontrado (búsqueda exitosa). Si  $x < a[m]$  se modifica  $j = m-1$ , sino se modifica  $i = m+1$  y se sigue iterando.

Implementación:

```

public int busquedaBinaria(int []a, int x)
{
    int i=0, j=a.length-1;
    while (i<=j)
    {
        int m=(i+j)/2;
        if (x==a[m])
        {
            return m;
        }
        else if (x<a[m])
        {
            j=m-1;
        }
        else
        {
            i=m+1;
        }
    }
    return NO_ENCONTRADO; // NO_ENCONTRADO se define como -1
}

```

## Eficiencia de la búsqueda binaria

Si la única operación permitida entre los elementos del conjunto es la comparación, entonces ¿qué tan eficiente es la búsqueda binaria?

Todo algoritmo de búsqueda basado en comparaciones corresponde a algún *árbol de decisión*. Cada nodo de dicho árbol corresponde al conjunto de elementos candidatos en donde se encuentra el elemento buscado, y que es consistente con las comparaciones realizadas entre los elementos. Los arcos del árbol corresponden a los resultados de las comparaciones, que en este caso pueden ser *mayor que* o *menor que* el elemento buscado, es decir, es un árbol de decisión binario.

El número de comparaciones realizadas por el algoritmo de búsqueda es igual a la altura del árbol de decisión (profundidad de la hoja más profunda).

Lema: sea  $D$  un árbol binario de altura  $h$ .  $D$  tiene a lo más  $2^h$  hojas.

Demostración: por inducción. Si  $h = 0$  el árbol tiene un solo nodo que necesariamente es una hoja (no tiene hijos), con lo que se tiene el caso base. En el caso general, se tiene una raíz, que no puede ser una hoja, que posee un subárbol izquierdo y derecho, cada uno con una altura máxima de  $h-1$ . Por hipótesis de inducción, los subárboles pueden tener a lo más  $2^{h-1}$  hojas, dando un total de a lo más  $2^h$  hojas entre ambos subárboles. Queda entonces demostrado.

Lema: un árbol binario con  $H$  hojas debe tener una profundidad de al menos  $\lceil \log_2(H) \rceil$

Demostración: directo del lema anterior.

Si  $n$  es el número de nodos de elementos del conjunto, el número de respuestas posibles (hojas del árbol de decisión) es de  $n+1$ , el lema anterior implica que el costo en el peor caso es mayor o igual que el logaritmo del número de respuestas posibles.



**Corolario:** cualquier algoritmo de búsqueda mediante comparaciones se demora al menos  $\lceil \log_2(n+1) \rceil$  preguntas en el peor caso. Por lo tanto, la búsqueda binaria es *óptima*.

## Búsqueda secuencial con probabilidades de acceso no uniforme

Se tiene un conjunto de elementos  $X_1, X_2, \dots, X_N$ , cuyas probabilidades de acceso son, respectivamente,  $P_1, P_2, \dots, P_N$ . Naturalmente:

$$\sum_{i=1}^N P_i = 1$$

El costo esperado de la búsqueda secuencial es:

$$Costo = \sum_{k=1}^N kP_k$$

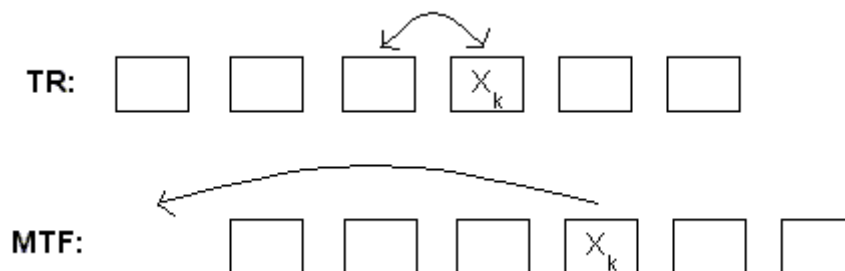
Este costo es *mínimo* cuando  $P_1 \geq P_2 \geq P_3 \dots \geq P_N$ .

¿Qué pasa si las probabilidades  $P_k$  **no** son conocidas de antemano? En este caso, no es posible ordenar *a priori* los elementos según su probabilidad de acceso de mayor a menor.

## Métodos auto-organizantes

*Idea:* cada vez que se accesa un elemento  $X_k$  se modifica la lista para que los accesos futuros a  $X_k$  sean más eficientes. Algunas políticas de modificación de la lista son:

- *TR (transpose):* se intercambia de posición  $X_k$  con  $X_{k-1}$  (siempre que  $k > 1$ ).
- *MTF (move-to-front):* se mueve el elemento  $X_k$  al principio de la lista.

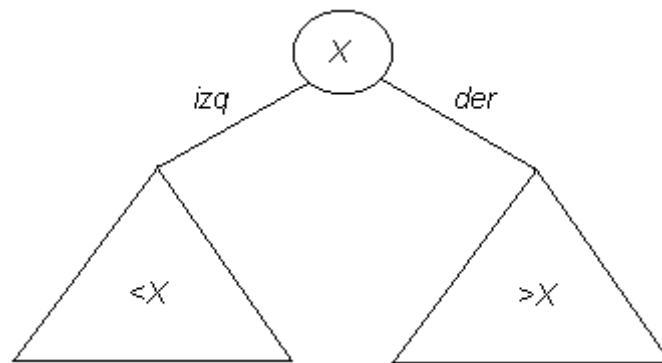


Se puede demostrar que  $Costo_{optimo} \leq Costo_{TR} \leq Costo_{MTF} \leq 2Costo_{optimo}$ .

## Arboles de búsqueda binaria

Un *árbol de búsqueda binaria*, en adelante *ABB*, es un árbol binario en donde todos los nodos cumplen la siguiente propiedad (sin perder generalidad se asumirá que los elementos almacenados son números enteros): si el valor del elemento almacenado en un nodo  $N$  es  $X$ , entonces todos los valores almacenados en el subárbol izquierdo de  $N$

son *menores* que  $X$ , y los valores almacenados en el subárbol derecho de  $N$  son *mayores* que  $X$ .



Los ABB permiten realizar de manera eficiente las operaciones provistas por el TDA diccionario, como se verá a continuación.

## Búsqueda en un ABB

Esta operación retorna una referencia al nodo en donde se encuentra el elemento buscado,  $X$ , o *null* si dicho elemento no se encuentra en el árbol. La estructura del árbol facilita la búsqueda:

- Si el árbol está vacío, entonces el elemento no está y se retorna *null*.
- Si el árbol no está vacío y el elemento almacenado en la raíz es  $X$ , se encontró el elemento y se retorna una referencia a dicho nodo.
- Si  $X$  es menor que el elemento almacenado en la raíz se sigue buscando recursivamente en el subárbol izquierdo, y si  $X$  es mayor que el elemento almacenado en la raíz se sigue buscando recursivamente en el subárbol derecho.

Nótese que el orden en los cuales se realizan los pasos anteriores es crucial para asegurar que la búsqueda en el ABB se lleve a cabo de manera correcta.

## Costo de búsqueda en un ABB

### Peor caso

Para un árbol dado, el peor caso es igual a la longitud del camino más largo desde la raíz a una hoja, y el peor caso sobre todos los árboles posibles es  $O(n)$ .

### Caso promedio

Supuestos:

- Árbol con  $n$  nodos.
- Probabilidad de acceso a los elementos uniforme.

Costo de búsqueda exitosa:

$$C_n = 1 + \frac{I_n}{n}$$

donde  $I_n$  es el largo de caminos internos.

Costo de búsqueda infructuosa:

$$C'_n = \frac{E_n}{n+1}$$

donde  $E_n$  es el largo de caminos externos.

Recordando que  $E_n = I_n + 2n$ , se tiene:

$$C_n = 1 + \frac{E_n - 2n}{n} = \frac{E_n}{n} - 1 = \frac{n+1}{n} C'_n - 1$$

Por lo tanto, la ecuación que relaciona los costos de búsqueda exitosa e infructuosa es:  
(\*)

$$C_n = \left(1 + \frac{1}{n}\right) C'_n - 1$$

Esto muestra que a medida que se insertan más elementos en el ABB los costos de búsqueda exitosa e infructuosa se van haciendo cada vez más parecidos.

El costo de búsqueda de un elemento en un ABB es igual al costo de búsqueda infructuosa justo antes de insertarlo más 1. Esto quiere decir que si ya habían  $k$  elementos en el árbol y se inserta uno más, el costo esperado de búsqueda para este último es  $1 + C'_k$ . Por lo tanto:

$$C_n = \frac{(1 + C'_0) + (1 + C'_1) + \dots + (1 + C'_{n-1})}{n} = 1 + \frac{1}{n} \sum_{k=0}^{n-1} C'_k$$

Esta última ecuación implica que:

$$\begin{aligned} nC_n &= n + \sum_{k=0}^{n-1} C'_k \\ (n+1)C_{n+1} &= n+1 + \sum_{k=0}^n C'_k \end{aligned}$$

Restando ambas ecuaciones se obtiene: (\*\*)

$$(n+1)C_{n+1} - nC_n = 1 + C'_n$$

De la ecuación (\*) se tiene:

$$nC_n = (n+1)C'_n - n$$

Reemplazando en (\*\*):

$$\begin{aligned}(n+2)C'_{n+1} - (n+1) - (n+1)C'_n + n &= 1 + C'_n \\ (n+2)(C'_{n+1} - C'_n) &= 2 \\ C'_{n+1} - C'_n &= \frac{2}{n+2}\end{aligned}$$

Obteniéndose la siguiente ecuación de recurrencia:

$$\begin{cases} C'_{n+1} = C'_n + \frac{2}{n+2} \\ C'_0 = 0 \end{cases}$$

Desenrollando la ecuación: (\*\*\*)

$$C'_n = \frac{2}{n+1} + C'_{n-1} = \frac{2}{n+1} + \frac{2}{n} + C'_{n-2} = \dots = 2 \cdot \left( \frac{1}{n+1} + \frac{1}{n} + \dots + \frac{1}{2} \right) + C'_0 = 2 \cdot \left( \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n+1} \right)$$

Se define  $H_n$ , conocido como *números armónicos*, como:

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n}$$

Se puede demostrar que:

$$\begin{aligned}H_n &\leq 1 + \int_1^n \frac{1}{x} dx = 1 + \ln(x) \Big|_1^n = 1 + \ln(n) \\ H_n &\geq \int_1^{n+1} \frac{1}{x} dx = \ln(x) \Big|_1^{n+1} = \ln(n+1) \\ \therefore \ln(n+1) &\leq H_n \leq 1 + \ln(n)\end{aligned}$$

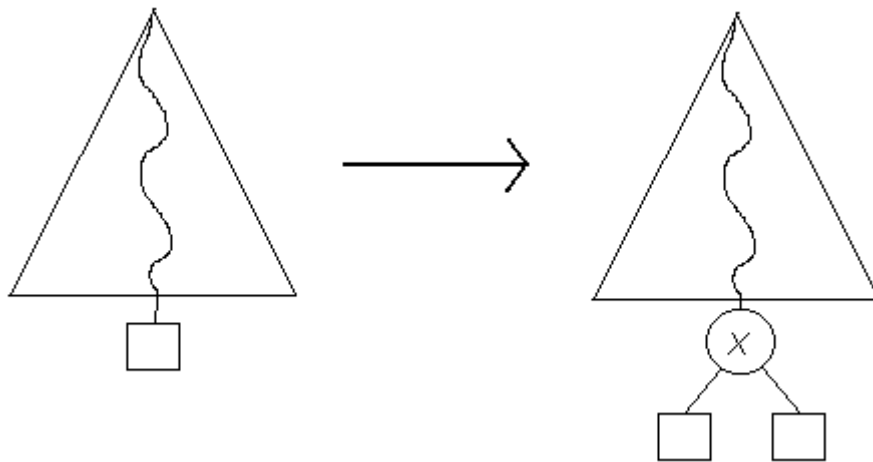
Reemplazando en (\*\*\*) y recordando (\*) se obtiene:

$$C_n \approx C'_n = 2(H_{n+1} - 1) \approx 2\ln(n) = 2\ln(2)\log_2(n) \approx 1.38\log_2(n) = O(\log_2 n)$$

Por lo tanto, el costo promedio de búsqueda en un ABB es  $O(\log(n))$ .

## Inserción en un ABB

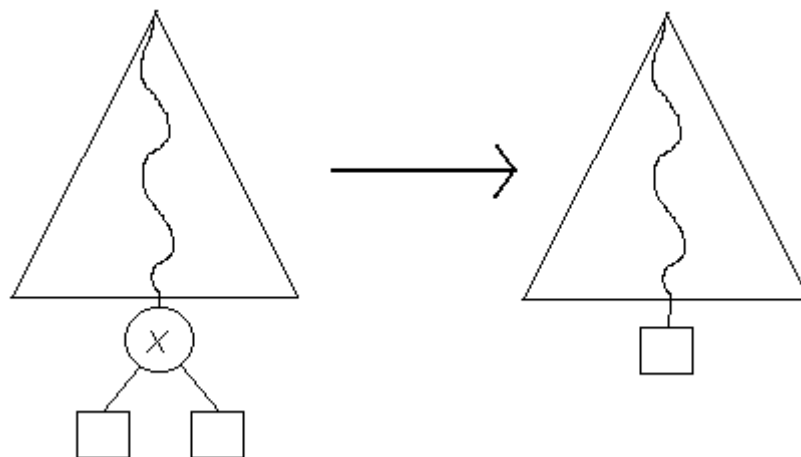
Para insertar un elemento  $X$  en un ABB, se realiza una búsqueda infructuosa de este elemento en el árbol, y en el lugar en donde debiera haberse encontrado se inserta. Como se vio en la sección anterior, el costo promedio de inserción en un ABB es  $O(\log(n))$ .



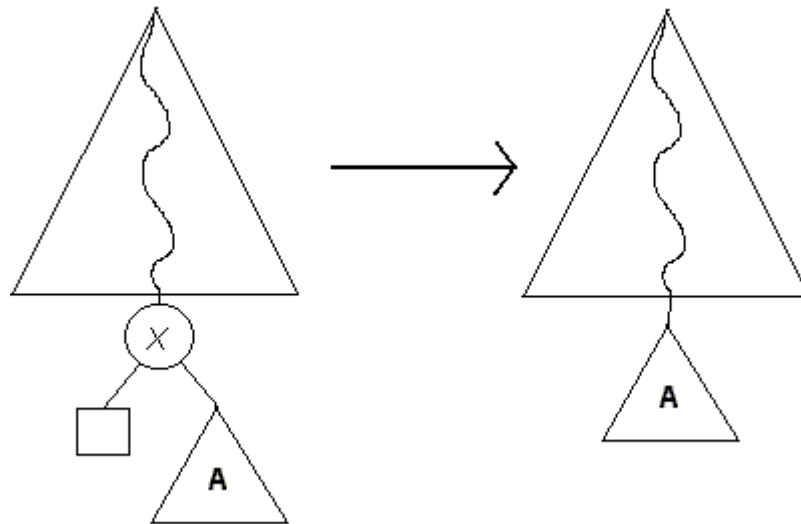
### Eliminación en un ABB

Primero se realiza una búsqueda del elemento a eliminar, digamos  $X$ . Si la búsqueda fue infructuosa no se hace nada, en caso contrario hay que considerar los siguientes casos posibles:

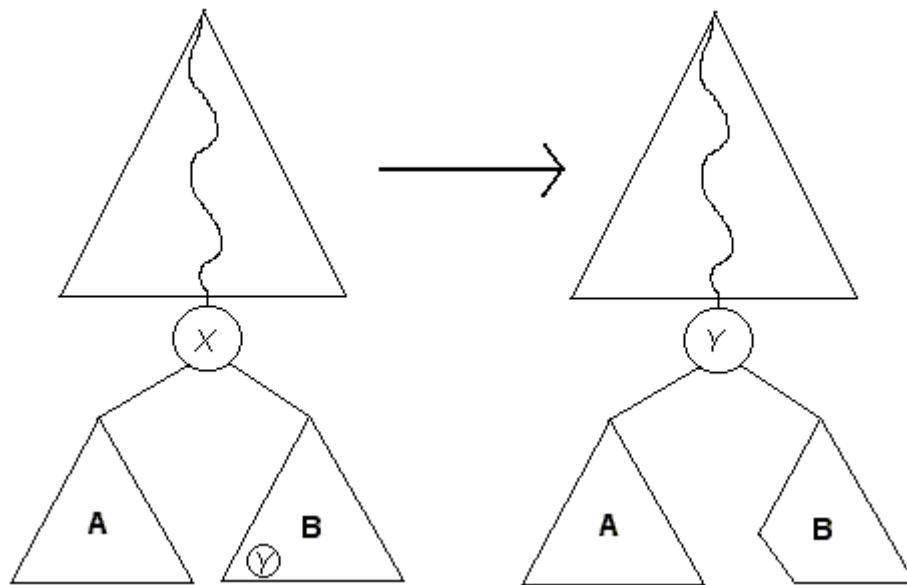
- Si  $X$  es una hoja sin hijos, se puede eliminar inmediatamente.



- Si  $X$  tiene un solo hijo, entonces se cambia la referencia del padre a  $X$  para que ahora referencie al hijo de  $X$ .



- Si  $X$  tiene dos hijos, el caso complicado, se procede de la siguiente forma: se elimina el nodo  $Y = \text{mínimo nodo del subárbol derecho de } X$ , el cual corresponde a uno de los casos fáciles de eliminación, y se reemplaza el valor almacenado en el nodo  $X$  por el valor que estaba almacenado en el nodo  $Y$ .



Nótese que el árbol sigue cumpliendo las propiedades de un ABB con este método de eliminación.

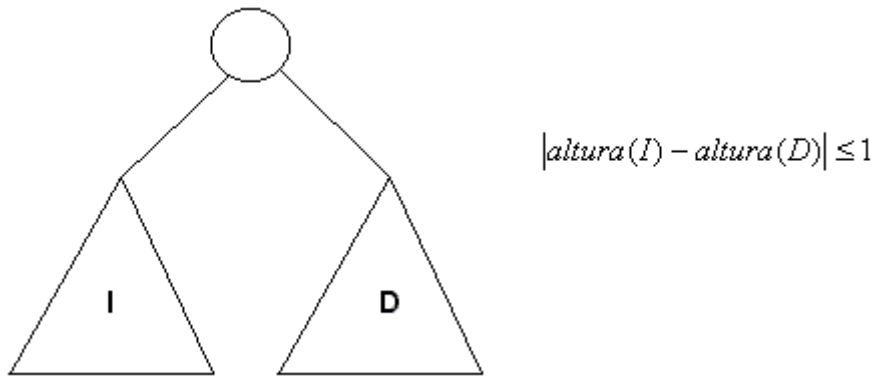
Si de antemano se sabe que el número de eliminaciones será pequeño, entonces la eliminación se puede substituir por una *marca* que indique si un nodo fue eliminado o no. Esta estrategia es conocida como *eliminación perezosa (lazy deletion)*.

## Arboles AVL

Definición: un *árbol balanceado* es un árbol que garantiza costos de búsqueda, inserción y eliminación en tiempo  $O(\log(n))$  incluso en el peor caso.

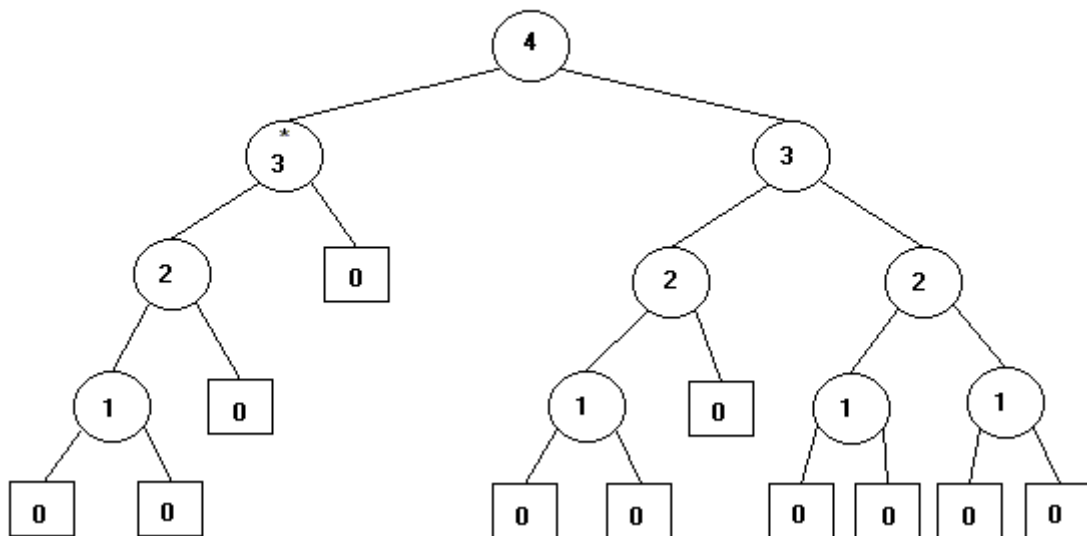
Un *árbol AVL* (Adelson-Velskii y Landis) es un árbol de búsqueda binaria que asegura un costo  $O(\log(n))$  en las operaciones de búsqueda, inserción y eliminación, es decir, posee una condición de balance.

La condición de balance es: *un árbol es AVL si para todo nodo interno la diferencia de altura de sus 2 árboles hijos es menor o igual que 1.*

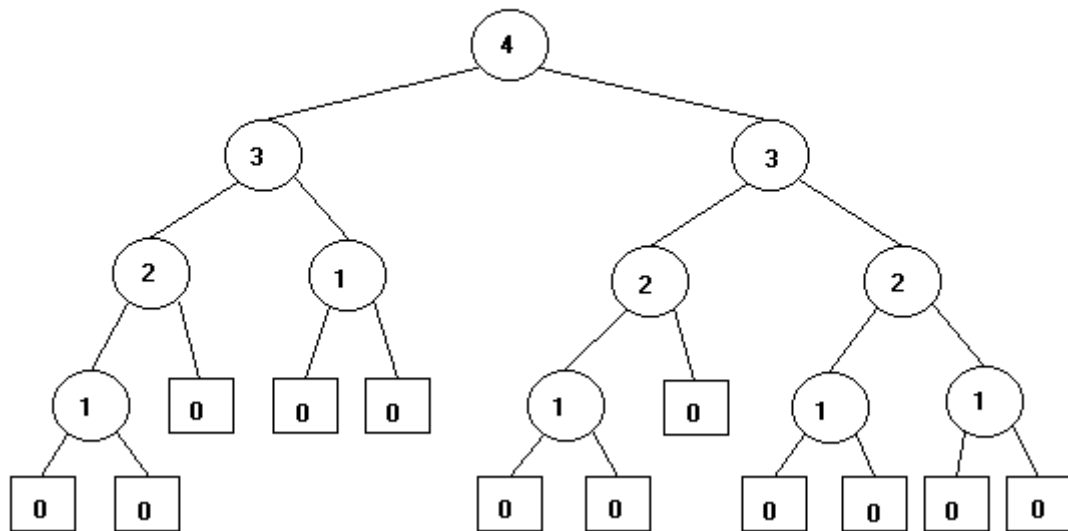


Por ejemplo: (el número dentro de cada nodo indica su altura)

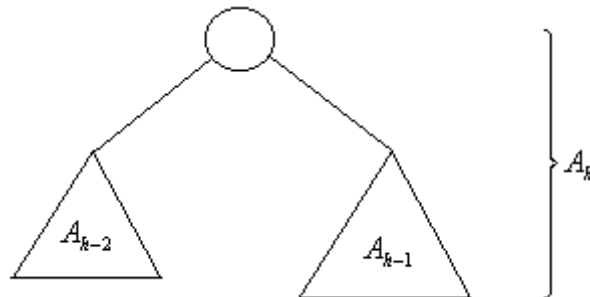
No es AVL (nodo \* no cumple condición)



Si es AVL



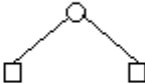
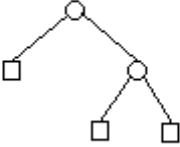
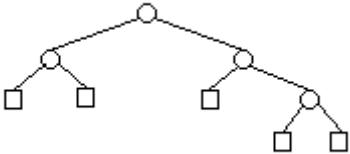
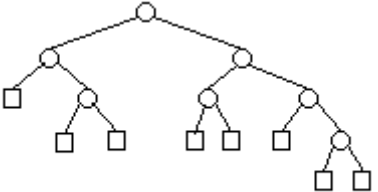
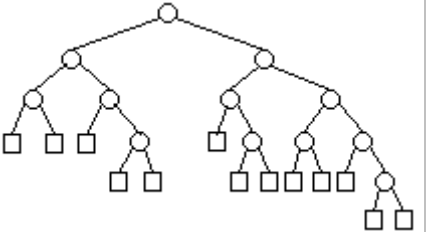
Problema: para una altura  $h$  dada, ¿cuál es el árbol AVL con mínimo número de nodos que alcanza esa altura?. Nótese que en dicho árbol AVL la diferencia de altura de sus hijos en todos los nodos tiene que ser 1 (demostrar por contradicción). Por lo tanto, si  $A_h$  representa al árbol AVL de altura  $h$  con mínimo número de nodos, entonces sus hijos deben ser  $A_{h-1}$  y  $A_{h-2}$ .



En la siguiente tabla  $n_h$  representa el número de nodos externos del árbol AVL con mínimo número de nodos internos.

$h$	$A_h$	$n_h$
0	□	1



1		2
2		3
3		5
4		8
5		13

Por inspección: ( $F_h$  representa los números de Fibonacci)

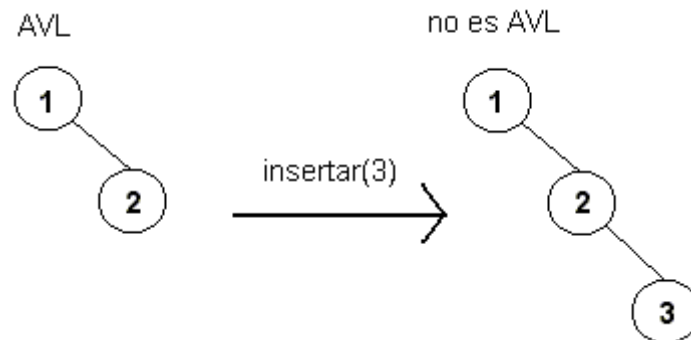
$$n_k = F_{k+2} \approx \frac{1}{\sqrt{5}} \phi^{k+2} \Rightarrow h \leq \log_{\phi}(n+1) + \Theta(1) = O(\log(n))$$

Por lo tanto, la altura de un árbol AVL es  $\Theta(\log(n))$ . Esto implica automáticamente que la búsqueda en un AVL tiene costo de  $\Theta(\log(n))$  en el peor caso.

## Inserción en un AVL

La inserción en un AVL se realiza de la misma forma que en un ABB, con la salvedad que hay que modificar la información de la altura de los nodos que se encuentran en el

camino entre el nodo insertado y la raíz del árbol. El problema potencial que se puede producir después de una inserción es que el árbol con el nuevo nodo no sea AVL:



En el ejemplo de la figura, la condición de balance se pierde al insertar el número 3 en el árbol, por lo que es necesario *restaurar* de alguna forma dicha condición. Esto siempre es posible de hacer a través de una modificación simple en el árbol, conocida como *rotación*.

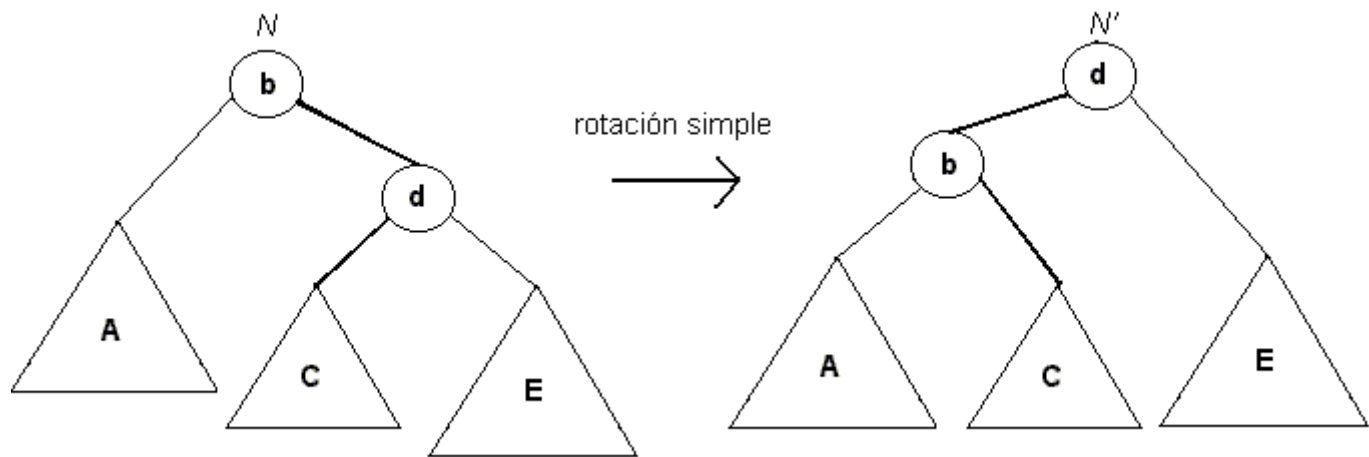
Suponga que después de la inserción de un elemento  $X$  el nodo desbalanceado más profundo en el árbol es  $N$ . Esto quiere decir que la diferencia de altura entre los dos hijos de  $N$  tiene que ser 2, puesto que antes de la inserción el árbol estaba balanceado. La violación del balance pudo ser ocasionada por alguno de los siguientes casos:

- El elemento  $X$  fue insertado en el subárbol izquierdo del hijo izquierdo de  $N$ .
- El elemento  $X$  fue insertado en el subárbol derecho del hijo izquierdo de  $N$ .
- El elemento  $X$  fue insertado en el subárbol izquierdo del hijo derecho de  $N$ .
- El elemento  $X$  fue insertado en el subárbol derecho del hijo derecho de  $N$ .

Dado que el primer y último caso son simétricos, así como el segundo y el tercero, sólo hay que preocuparse de dos casos principales: una inserción "hacia afuera" con respecto a  $N$  (primer y último caso) o una inserción "hacia adentro" con respecto a  $N$  (segundo y tercer caso).

### **Rotación simple**

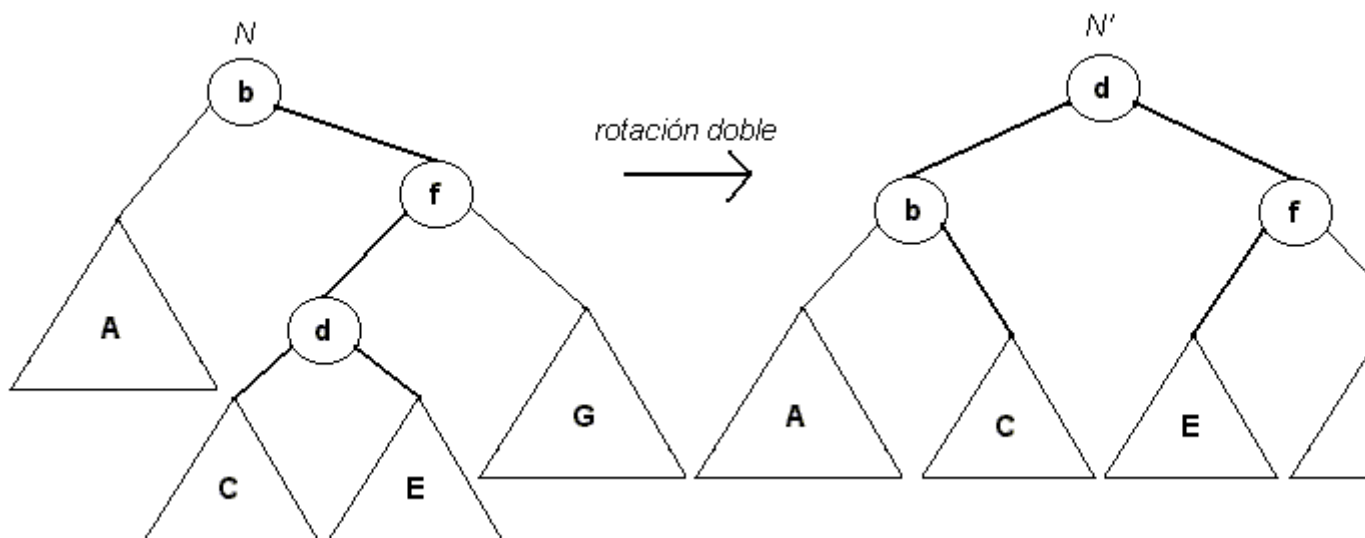
El desbalance por inserción "hacia afuera" con respecto a  $N$  se soluciona con una *rotación simple*.



La figura muestra la situación antes y después de la rotación simple, y ejemplifica el cuarto caso anteriormente descrito, es decir, el elemento  $X$  fue insertado en  $E$ , y  $b$  corresponde al nodo  $N$ . Antes de la inserción, la altura de  $N$  es la altura de  $C+1$ . Idealmente, para recuperar la condición de balance se necesitaría *bajar A* en un nivel y *subir E* en un nivel, lo cual se logra cambiando las referencias *derecha* de  $b$  e *izquierda* de  $d$ , quedando este último como nueva raíz del árbol,  $N'$ . Nótese que ahora el nuevo árbol *tiene la misma altura que antes de insertar el elemento,  $C+1$* , lo cual implica que *no puede haber nodos desbalanceados más arriba en el árbol*, por lo que es necesaria una sola rotación simple para devolver la condición de balance al árbol. Nótese también que el árbol *sigue cumpliendo con la propiedad de ser ABB*.

### Rotación doble

Claramente un desbalance producido por una inserción "hacia adentro" con respecto a  $N$  no es solucionado con una rotación simple, dado que ahora es  $C$  quien produce el desbalance y como se vio anteriormente este subárbol mantiene su posición relativa con una rotación simple.



Para el caso de la figura (tercer caso), la altura de  $N$  antes de la inserción era  $G+1$ . Para recuperar el balance del árbol es necesario *subir C y E* y *bajar A*, lo cual se logra realizando *dos rotaciones simples: la primera entre d y f*, y la *segunda entre d*, ya

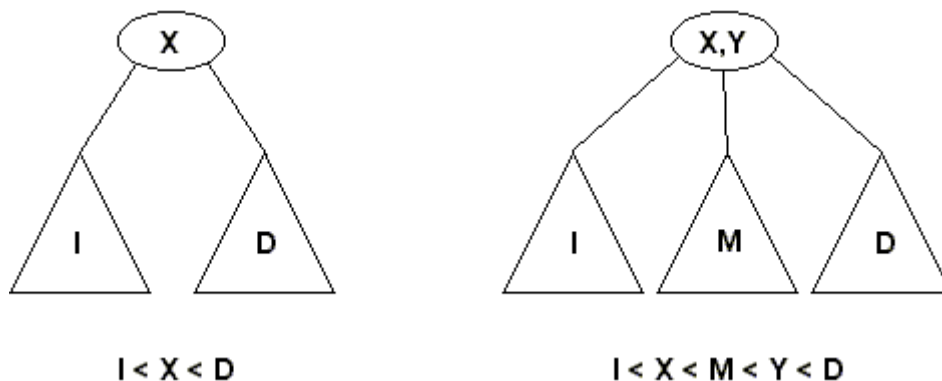
*rotado*, y **b**, obteniéndose el resultado de la figura. A este proceso de dos rotaciones simples se le conoce como *rotación doble*, y como la altura del nuevo árbol  $N'$  es la misma que antes de la inserción del elemento, ningún elemento hacia arriba del árbol queda desbalanceado, por lo que solo es necesaria una rotación doble para recuperar el balance del árbol después de la inserción. Nótese que el nuevo árbol cumple con la propiedad de ser ABB después de la rotación doble.

## Eliminación en un AVL

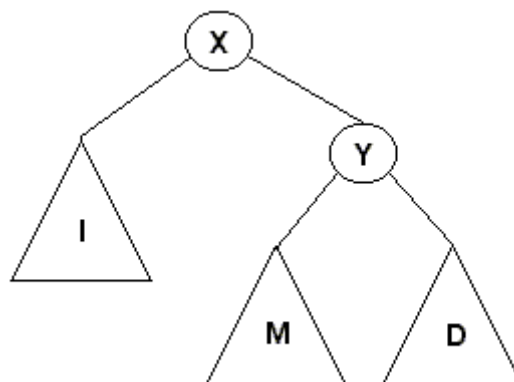
La eliminación en árbol AVL se realiza de manera análoga a un ABB, pero también es necesario verificar que la condición de balance se mantenga una vez eliminado el elemento. En caso que dicha condición se pierda, será necesario realizar una rotación simple o doble dependiendo del caso, pero es posible que se requiera más de una rotación para reestablecer el balance del árbol.

## Arboles 2-3

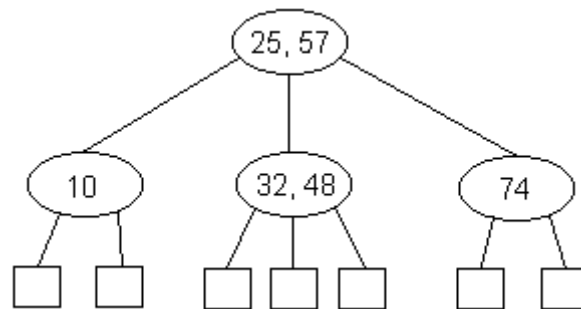
Los *árboles 2-3* son árboles cuyos nodos internos pueden contener *hasta 2 elementos* (todos los árboles vistos con anterioridad pueden contener sólo un elemento por nodo), y por lo tanto un nodo interno puede tener 2 o 3 hijos, dependiendo de cuántos elementos posea el nodo. De este modo, un nodo de un árbol 2-3 puede tener una de las siguientes formas:



Un árbol 2-3 puede ser simulado utilizando árboles binarios:



Una propiedad de los árboles 2-3 es que *todas las hojas están a la misma profundidad*, es decir, los árboles 2-3 son árboles *perfectamente balanceados*. La siguiente figura muestra un ejemplo de un árbol 2-3:



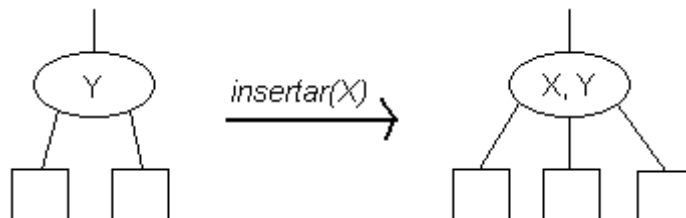
Nótese que se sigue cumpliendo la propiedad de los árboles binarios: *nodos internos + 1 = nodos externos*. Dado que el árbol 2-3 es perfectamente balanceado, la altura de éste está acotada por:

$$\log_3(n) \leq \text{altura} \leq \log_2(n)$$

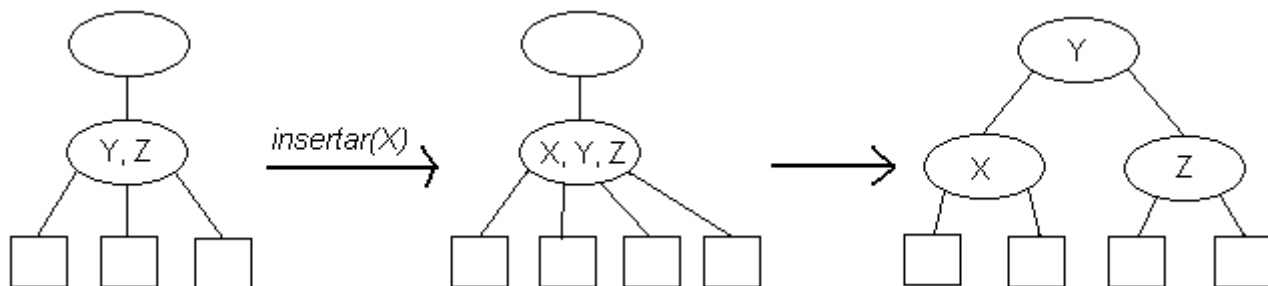
### Inserción en un árbol 2-3

Para insertar un elemento  $X$  en un árbol 2-3 se realiza una búsqueda infructuosa y se inserta dicho elemento en el último nodo visitado durante la búsqueda, lo cual implica manejar dos casos distintos:

- Si el nodo donde se inserta  $X$  tenía una sola llave (dos hijos), ahora queda con dos llaves (tres hijos).



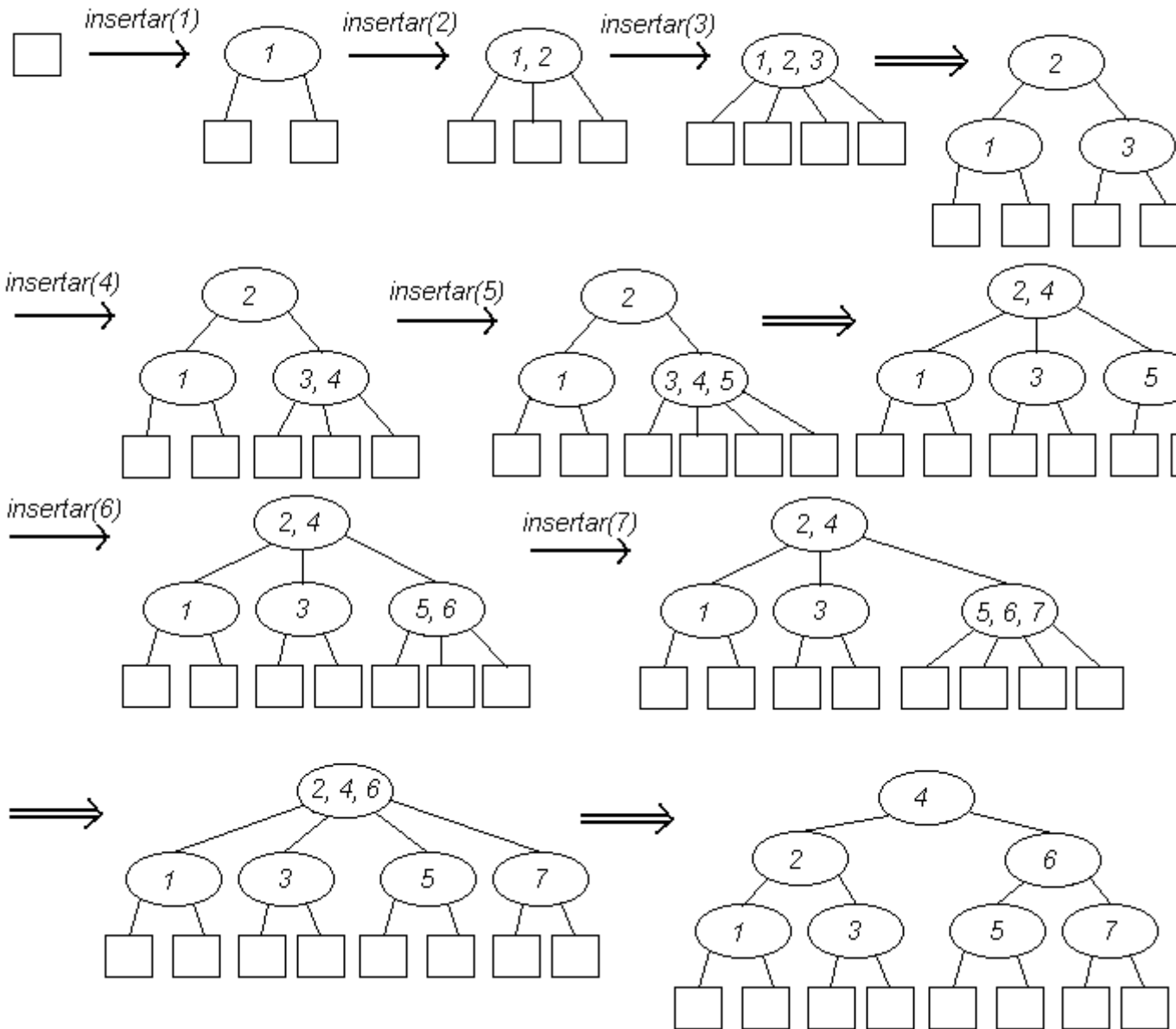
- Si el nodo donde se inserta  $X$  tenía dos llaves (tres hijos), queda transitoriamente con tres llaves (cuatro hijos) y se dice que está *saturado (overflow)*.



El problema se resuelve a nivel de  $X$  y  $Z$ , pero es posible que el nodo que contiene a  $Y$  ahora este saturado. En este caso, se repite el mismo procedimiento

anterior un nivel más arriba. Finalmente, si la raíz es el nodo saturado, éste se divide y se crea una nueva raíz un nivel más arriba. Esto implica que los árboles 2-3 crecen "hacia arriba".

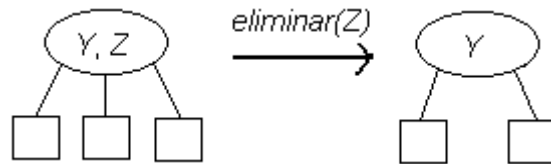
Ejemplos de inserción en árboles 2-3:



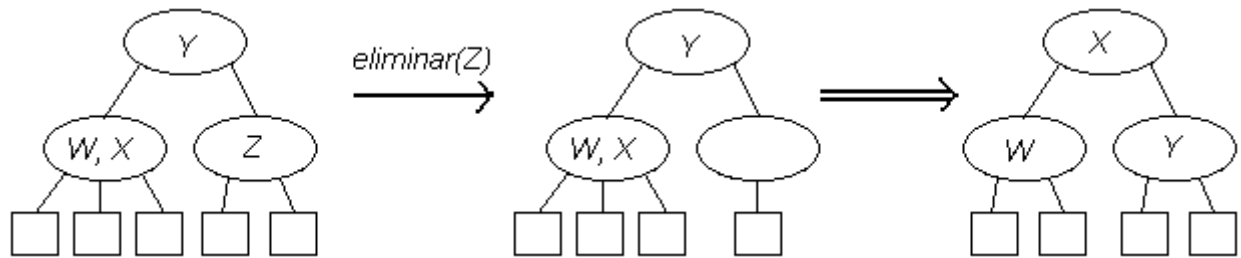
## Eliminación en un árbol 2-3

Sin perder generalidad se supondrá que el elemento a eliminar,  $Z$ , se encuentra en el nivel más bajo del árbol. Si esto no es así, entonces el sucesor y el predecesor de  $Z$  se encuentran necesariamente en el nivel más bajo (¿por qué?); en este caso basta con borrar uno de ellos y luego escribir su valor sobre el almacenado en  $Z$ . La eliminación también presenta dos posibles casos:

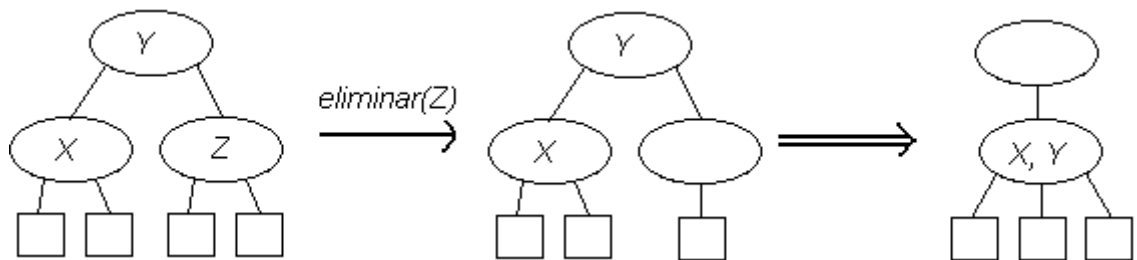
- El nodo donde se encuentra  $Z$  contiene dos elementos. En este caso se elimina  $Z$  y el nodo queda con un solo elemento.



- El nodo donde se encuentra Z contiene un solo elemento. En este caso al eliminar el elemento Z el nodo queda sin elementos (*underflow*). Si el nodo hermano posee dos elementos, se le quita uno y se inserta en el nodo con *underflow*.



Si el nodo hermano contiene solo una llave, se le quita un elemento al padre y se inserta en el nodo con *underflow*.



Si esta operación produce *underflow* en el nodo padre, se repite el procedimiento anterior un nivel más arriba. Finalmente, si la raíz queda vacía, ésta se elimina.

Costo de las operaciones de búsqueda, inserción y eliminación en el peor caso:  
 $\Theta(\log(n))$ .

## Arboles B

La idea de los árboles 2-3 se puede generalizar a árboles  $t - (2t-1)$ , donde  $t \geq 2$  es un parámetro fijo, es decir, cada nodo del árbol posee entre  $t$  y  $2t-1$  hijos, excepto por la raíz que puede tener entre 2 y  $2t-1$  hijos. En la práctica,  $t$  puede ser bastante grande, por ejemplo  $t = 100$  o más. Estos árboles son conocidos como *árboles B* (Bayer).

### Inserción en un árbol B

- Se agrega una nueva llave al nivel inferior.
- Si el nodo rebalsa (*overflow*), es decir, si queda con  $2t$  hijos, se divide en 2 nodos con  $t$  hijos cada uno y sube un elemento al nodo padre. Si el padre rebalsa, se repite el procedimiento un nivel más arriba.

- Finalmente, si la raíz rebalsa, se divide en 2 y se crea una nueva raíz un nivel más arriba.

## Eliminación en un árbol B

- Se elimina la llave, y su hoja respectiva, del nivel inferior.
- Si el nodo queda con menos de  $t$  hijos (underflow) se le piden prestados algunos al hermano, por ejemplo mitad y mitad.
- Si el hermano no tiene para prestar, entonces se fusionan los 2 nodos y se repite el procedimiento con el padre.
- Si la raíz queda vacía, se elimina.

## Variantes de los árboles B

- *Arboles  $B^*$* : cuando un nodo rebalsa se trasladan hijos hacia el hermano, y sólo se crea un nuevo nodo cuando ambos rebalsan. Esto permite aumentar la utilización mínima de los nodos, que antes era de un 50%.
- *Arboles  $B^+$* : La información solo se almacena en las hojas, y los nodos internos contienen los separadores que permiten realizar la búsqueda de elementos.

## Arboles digitales

Suponga que los elementos de un conjunto se pueden representar como una secuencia de *bits*:

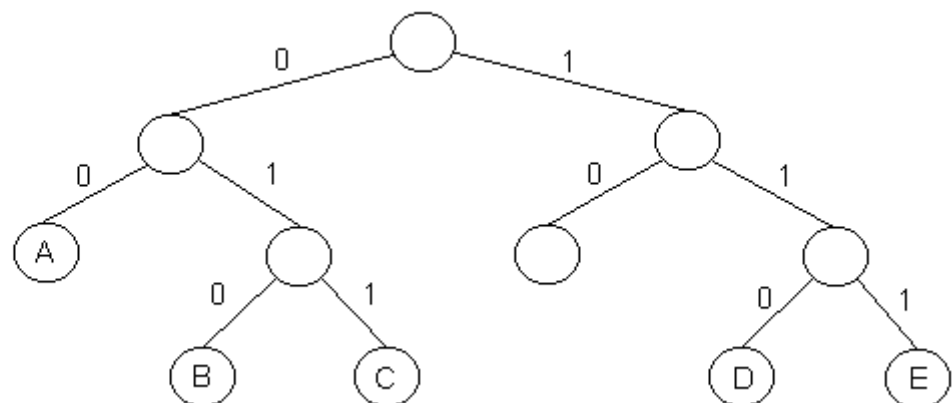
$$X = b_0b_1b_2...b_k$$

Además, suponga que ninguna representación de un elemento en particular es *prefijo* de otra. Un *árbol digital* es un árbol binario en donde la posición de inserción de un elemento ya no depende de su valor, sino de su representación binaria. Los elementos en un árbol digital se almacenan *sólo en sus hojas*, pero no necesariamente todas las hojas contienen elementos (ver ejemplo más abajo). Esta estructura de datos también es conocida como *trie*.

El siguiente ejemplo muestra un árbol digital con 5 elementos.

Codificación:

A = 00100  
B = 01000  
C = 01111  
D = 11000  
E = 11101





## Búsqueda en un árbol digital

Para buscar un elemento  $X$  en un árbol digital se procede de la siguiente manera:

- Se examinan los bits  $b_i$  del elemento  $X$ , partiendo desde  $b_0$  en adelante.
- Si  $b_i = 0$  se avanza por la rama izquierda y se examina el siguiente bit,  $b_{i+1}$ .
- Si  $b_i = 1$  se avanza por la rama derecha y se examina el siguiente bit.
- El proceso termina cuando se llega a una hoja, único lugar posible en donde puede estar insertado  $X$ .

## Inserción en un árbol digital

Suponga que se desea insertar el elemento  $X$  en el árbol digital. Se realiza una búsqueda infructuosa de  $X$  hasta llegar a una hoja, suponga que el último bit utilizado en la búsqueda fue  $b_k$ . Si la hoja está vacía, se almacena  $X$  en dicha hoja. En caso contrario, se divide la hoja utilizando el siguiente bit del elemento,  $b_{k+1}$ , y se repite el procedimiento, si es necesario, hasta que quede sólo un elemento por hoja.

Con este proceso de inserción, la forma que obtiene el árbol digital es *insensible al orden de inserción de los elementos*.

## Eliminación en un árbol digital

Suponga que el elemento a eliminar es  $X$ . Se elimina el elemento de la hoja, y por lo tanto ésta queda *vacía*. Si la hoja vacía es *hermana* de otra hoja *no vacía*, entonces ambas se *fusionan* y se repite el procedimiento mientras sea posible.

## Costo promedio de búsqueda en un árbol digital

El costo promedio de búsqueda en un árbol digital es mejor que en un ABB, ya que en un árbol digital la probabilidad que un elemento se encuentre en el subárbol izquierdo o derecho es la misma,  $1/2$ , dado que sólo depende del valor de un bit (0 o 1). En cambio, en un ABB dicha probabilidad es proporcional al "peso" del subárbol respectivo.

$$\frac{H_n}{\ln(2)} + \frac{1}{2} + P(\log(n)) \approx \log_2 n + \frac{\gamma}{\ln(2)} + \frac{1}{2} = O(\log(n))$$

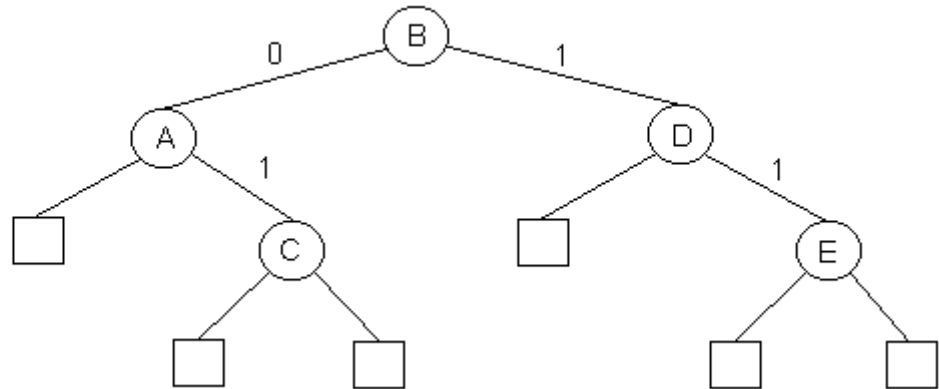
$H_n$  son los números armónicos y  $P(n)$  es una función periódica de muy baja amplitud ( $O(10^{-6})$ )

## Arboles de búsqueda digital

En este tipo de árboles los elementos se almacenan en los *nodos internos*, al igual que en un ABB, pero la ramificación del árbol es según los bits de las llaves. El siguiente ejemplo muestra un *árbol de búsqueda digital (ABD)*, en donde el orden de inserción es B, A, C, D, E:

Codificación:

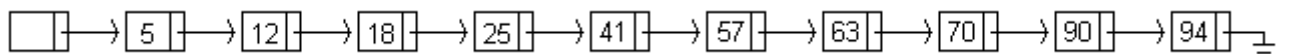
A = 00100  
B = 01000  
C = 01111  
D = 11000  
E = 11101



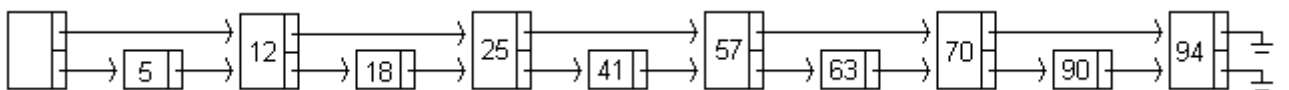
Los ABD poseen un mejor costo promedio de búsqueda que los ABB, pero también es  $O(\log(n))$ .

## Skip lists

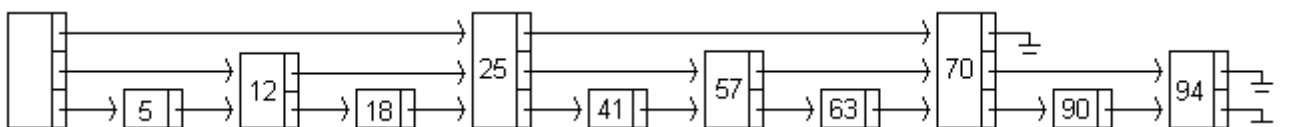
Al principio del capítulo se vio que una de las maneras simples de implementar el TDA diccionario es utilizando una lista enlazada, pero también se vio que el tiempo de búsqueda promedio es  $O(n)$  cuando el diccionario posee  $n$  elementos. La figura muestra un ejemplo de lista enlazada simple con cabecera, donde los elementos están ordenados ascendentemente:



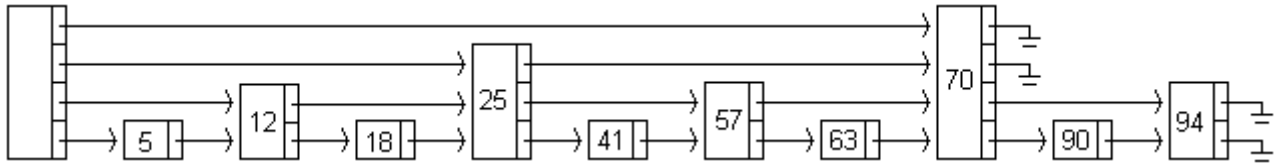
Sin embargo, es posible modificar la estructura de datos de la lista y así mejorar el tiempo de búsqueda promedio. La siguiente figura muestra una lista enlazada en donde a cada nodo par se le agrega una referencia al nodo ubicado dos posiciones más adelante en la lista. Modificando ligeramente el algoritmo de búsqueda, a lo más  $\lceil n/2 \rceil + 1$  nodos son examinados en el peor caso.



Esta idea se puede extender agregando una referencia cada cuatro nodos. En este caso, a lo más  $\lceil n/4 \rceil + 2$  nodos son examinados:

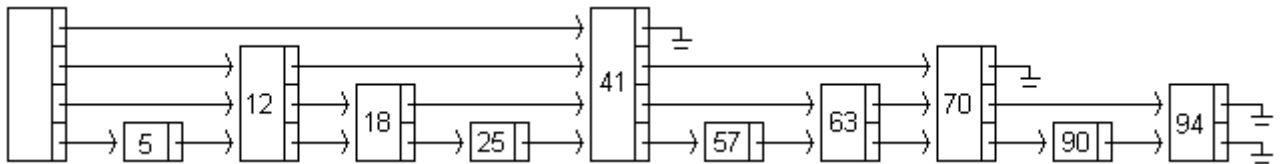


El caso límite para este argumento se muestra en la siguiente figura. Cada  $2^i$  nodo posee una referencia al nodo  $2^i$  posiciones más adelante en la lista. El número total de referencias solo ha sido doblado, pero ahora a lo más  $\lceil \log_2(n) \rceil$  nodos son examinados durante la búsqueda. Note que la búsqueda en esta estructura de datos es básicamente una búsqueda binaria, por lo que el tiempo de búsqueda en el peor caso es  $O(\log n)$ .



El problema que tiene esta estructura de datos es que es demasiado rígida para permitir inserciones de manera eficiente. Por lo tanto, es necesario *relajar* levemente las condiciones descritas anteriormente para permitir inserciones eficientes.

Se define un *nodo de nivel k* como aquel nodo que posee *k* referencias. Se observa de la figura anterior que, aproximadamente, la mitad de los nodos son de nivel 1, que un cuarto de los nodos son de nivel 2, etc. En general, aproximadamente  $n/2^i$  nodos son de nivel *i*. Cada vez que se inserta un nodo, se elige el nivel que tendrá aleatoriamente en concordancia con la distribución de probabilidad descrita. Por ejemplo, se puede lanzar una moneda al aire, y mientras salga *cara* se aumenta el nivel del nodo a insertar en 1 (partiendo desde 1). Esta estructura de datos es denominada *skip list*. La siguiente figura muestra un ejemplo de una skip list:



## Búsqueda en una skip list

Suponga que el elemento a buscar es *X*. Se comienza con la referencia superior de la cabecera. Se realiza un recorrido a través de este nivel (pasos horizontales) hasta que el siguiente nodo sea mayor que *X* o sea *null*. Cuando esto ocurra, se continúa con la búsqueda pero un nivel más abajo (paso vertical). Cuando no se pueda bajar de nivel, esto es, ya se está en el nivel inferior, entonces se realiza una comparación final para saber si el elemento *X* está en la lista o no.

## Inserción en una skip list

Se procede como en la búsqueda, manteniendo una marca en los nodos donde se realizaron pasos verticales, hasta llegar al nivel inferior. El nuevo elemento se inserta en la posición en donde debiera haberse encontrado, se calcula aleatoriamente su nivel y se actualizan las referencias de los nodos marcados dependiendo del nivel del nuevo nodo de la lista.

## Costo de búsqueda en una skip list

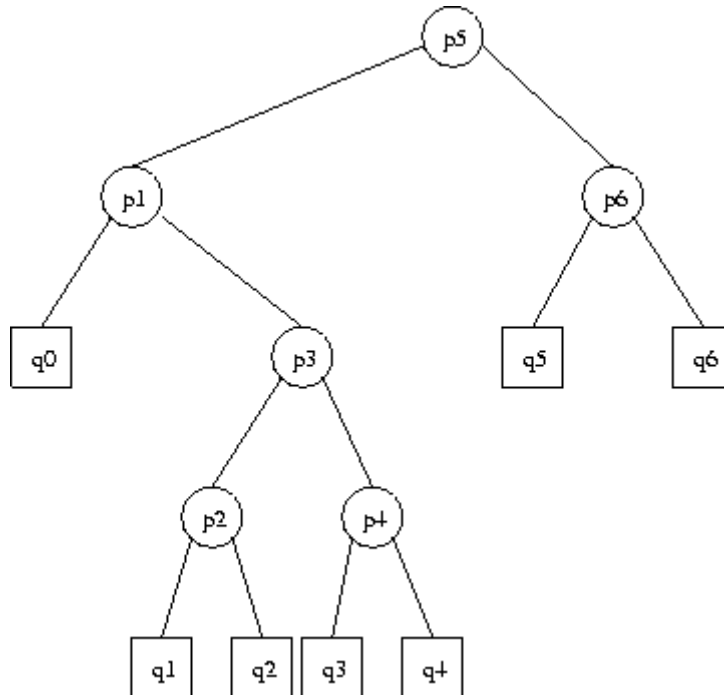
El análisis del costo de búsqueda promedio en una skip list es complicado, pero se puede demostrar que en promedio es  $O(\log n)$ . También se puede demostrar que si se usa una moneda cargada, es decir, que la probabilidad que salga cara es *p* y la probabilidad que salga sello es  $q = 1-p$ , entonces el costo de búsqueda es mínimo para  $p = 1/e$  (*e* = base del logaritmo natural).

# ABB óptimos

## ABB con probabilidades de acceso no uniforme

Problema: dados  $n$  elementos  $X_1 < X_2 < \dots < X_n$ , con probabilidades de acceso conocidas  $p_1, p_2, \dots, p_n$ , y con probabilidades de búsqueda infructuosa conocidas  $q_0, q_1, \dots, q_n$ , se desea encontrar el ABB que minimice el costo esperado de búsqueda.

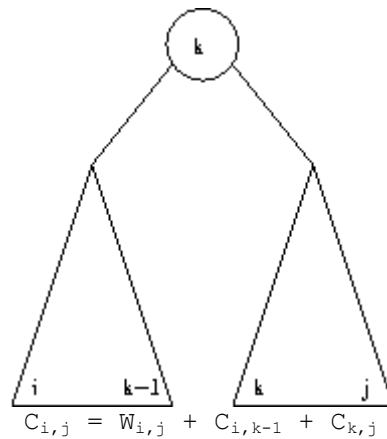
Por ejemplo, para el siguiente ABB con 6 elementos:



El costo esperado de búsqueda es:

$$\begin{aligned} C(q_0, q_6) &= (2q_0 + 2p_1 + 4q_1 + 4p_2 + 4q_2 + 3p_3 + 4q_3 + 4p_4 + 4q_4) + p_5 + (2q_5 + 2p_6 + 2q_6) \\ &= (q_0 + p_1 + q_1 + p_2 + q_2 + p_3 + q_3 + p_4 + q_4 + p_5 + q_5 + p_6 + q_6) + \\ &\quad (1q_0 + 1p_1 + 3q_1 + 3p_2 + 3q_2 + 2p_3 + 3q_3 + 3p_4 + 3q_4) + \\ &\quad (1q_5 + 1p_6 + 1q_6) \\ &= W(q_0, q_6) + C(q_0, q_4) + C(q_5, q_6) \end{aligned}$$

Esto es, el costo del árbol completo es igual al "peso" del árbol más los costos de los subárboles. Si la raíz es  $k$ :



Si el árbol completo es óptimo, entonces los subárboles también lo son, pero al revés no necesariamente es cierto, porque la raíz  $k$  puede haberse escogido mal. Luego, para encontrar el verdadero costo óptimo  $C_{opt_{i,j}}$  es necesario probar con todas las maneras posibles de elegir la raíz  $k$ .

$$C_{opt_{i,j}} = \min_{i+1 \leq k \leq j} \{W_{i,j} + C_{opt_{i,k-1}} + C_{opt_{k,j}}\}$$

$$C_{opt_{i,i}} = 0 \text{ para todo } i=0..n$$

Note que el "peso"  $W_{i,j}$  no depende de la variable  $k$ .

## Implementación de ABB óptimos

- Recursiva (equivalente a fuerza bruta): Calcular todos los árboles posibles ( $O(4^n)$ ).
  - Usando programación dinámica (tabulación):
- ```

for (i=0; i<=n; i++)
{
    C[i,i]=0; // subarboles de tamaño 0
    W[i,i]=q_i;
}
for (l=1; l<=n; l++)
    for (i=0; i<=n-l; i++)
    {
        j=i+l;
        W[i,j]=W[i,j-1]+p_j+q_j;
        C[i,j]=min_{i+1<=k<=j} {W[i,j]+C[i,k-1]+C[k,j]}
    }

```

Tiempo:  $O(n^3)$ .

Una mejora: se define  $r_{i,j}$  como el  $k$  que minimiza  $\min_{i+1 \leq k \leq j} \{W[i,j]+C[i,k-1]+C[k,j]\}$ . Intuitivamente  $r_{i,j-1} \leq r_{i,j} \leq r_{i+1,j}$ , y como  $r_{i,j-1}$  y  $r_{i+1,j}$  ya son conocidos al momento de calcular  $r_{i,j}$ , basta con calcular  $\min_{r_{i,j-1} \leq k \leq r_{i+1,j}} \{W[i,j]+C[i,k-1]+C[k,j]\}$ .

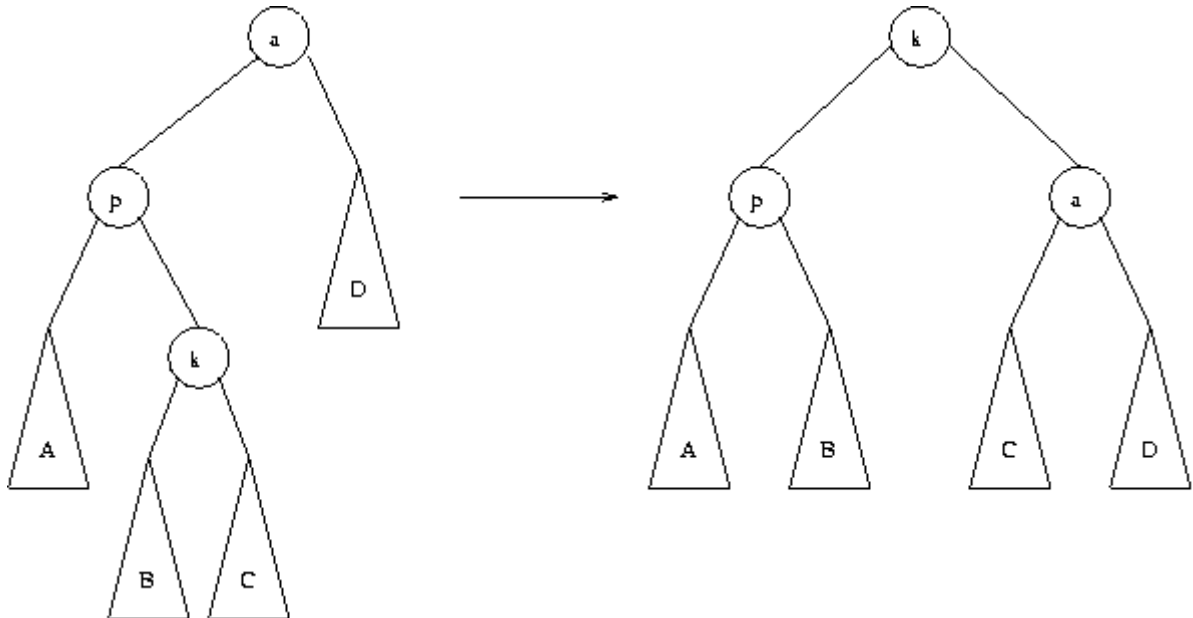
Con esta mejora, se puede demostrar que el método demora  $O(n^2)$  (Ejercicio: demostrarlo).

# Splay Trees

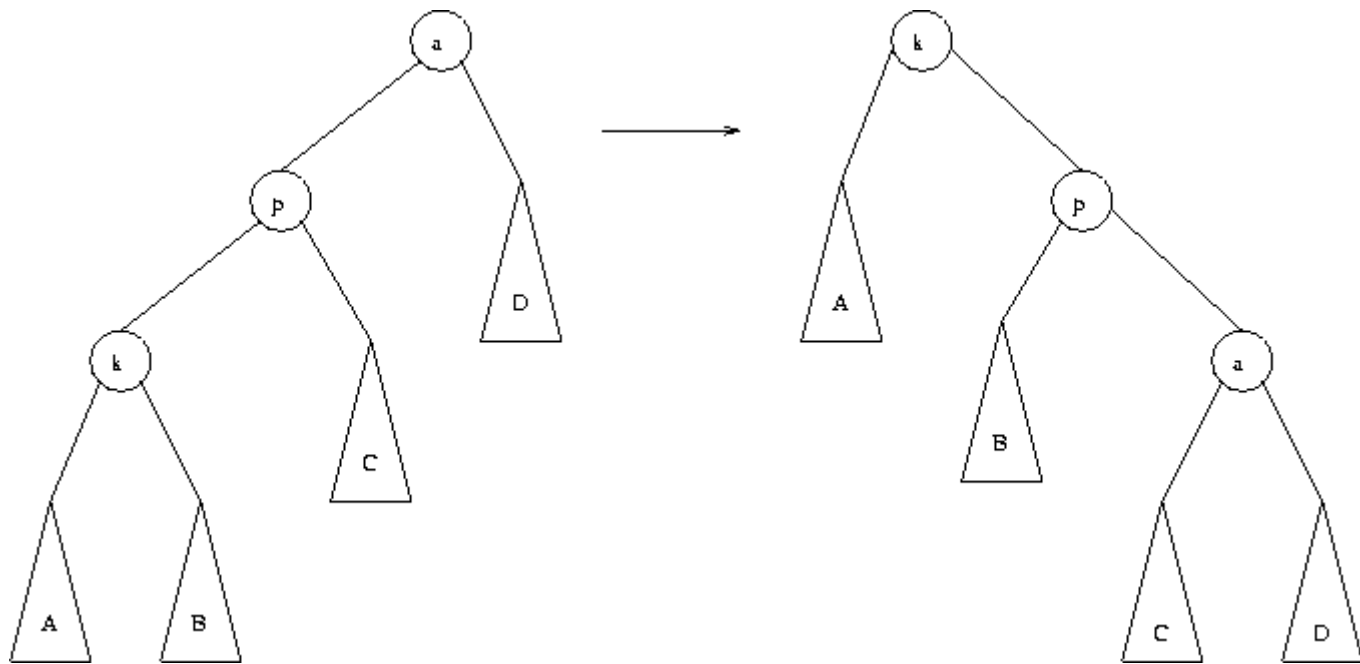
Esta estructura garantiza que para cualquier secuencia de  $M$  operaciones en un árbol, empezando desde un árbol vacío, toma a lo más tiempo  $O(M \log(N))$ . A pesar que esto no garantiza que alguna operación en particular tome tiempo  $O(N)$ , si asegura que no existe ninguna secuencia de operaciones que sea mala. En general, cuando una secuencia de  $M$  operaciones toma tiempo  $O(M f(N))$ , se dice que el costo *amortizado* en tiempo de cada operación es  $O(f(N))$ . Por lo tanto, en un splay tree los costos amortizados por operación son de  $O(\log(N))$ .

La idea básica de un splay tree es que después que un nodo es accesado éste se "sube" hasta la raíz del árbol a través de rotaciones al estilo AVL. Una manera de realizar esto, que NO funciona, es realizar rotaciones simples entre el nodo accesado y su padre hasta dejar al nodo accesado como raíz del árbol. El problema que tiene este enfoque es que puede dejar otros nodos muy abajo en el árbol, y se puede probar que existe una secuencia de  $M$  operaciones que toma tiempo  $O(M N)$ , por lo que esta idea no es muy buena.

La estrategia de "splaying" es similar a la idea de las rotaciones simples. Si el nodo  $k$  es accesado, se realizarán rotaciones para llevarlo hasta la raíz del árbol. Sea  $k$  un nodo distinto a la raíz del árbol. Si el padre de  $k$  es la raíz del árbol, entonces se realiza una rotación simple entre estos dos nodos. En caso contrario, el nodo  $k$  posee un nodo padre  $p$  y un nodo "abuelo"  $a$ . Para realizar las rotaciones se deben considerar dos casos posibles (más los casos simétricos).



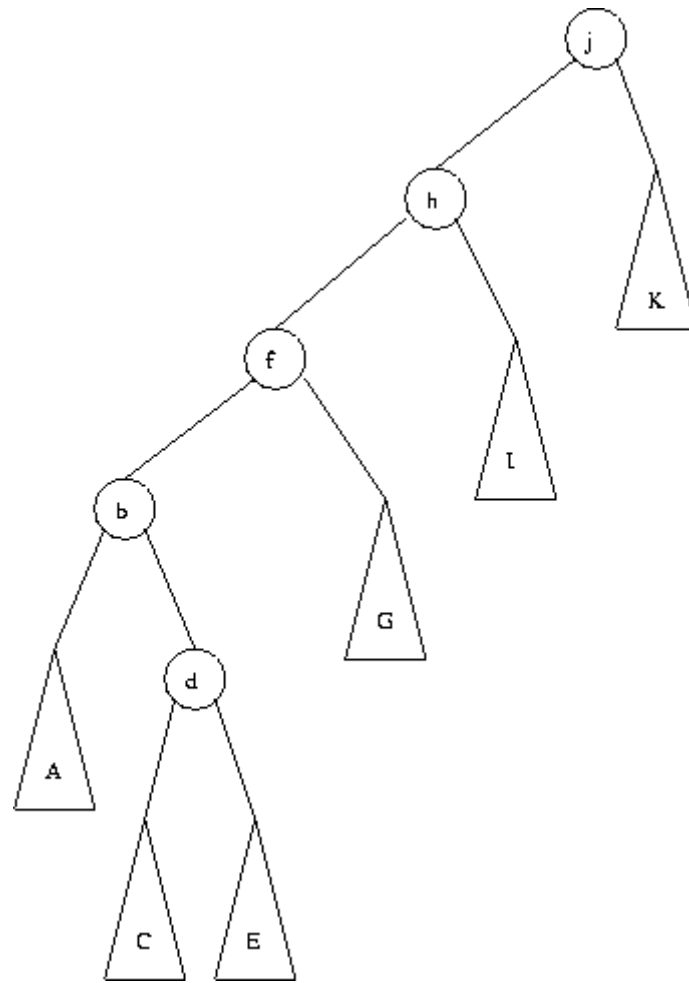
El primer caso es una inserción *zig-zag*, en donde  $k$  es un hijo derecho y  $p$  es un hijo izquierdo (o viceversa). En este caso, se realiza una rotación doble estilo AVL (ver figura superior).



El otro caso es una inseción *zig-zig*, en donde  $k$  y  $p$  son ambos hijos izquierdo o derecho. En este caso, se realiza la transformación indicada en la figura anterior.

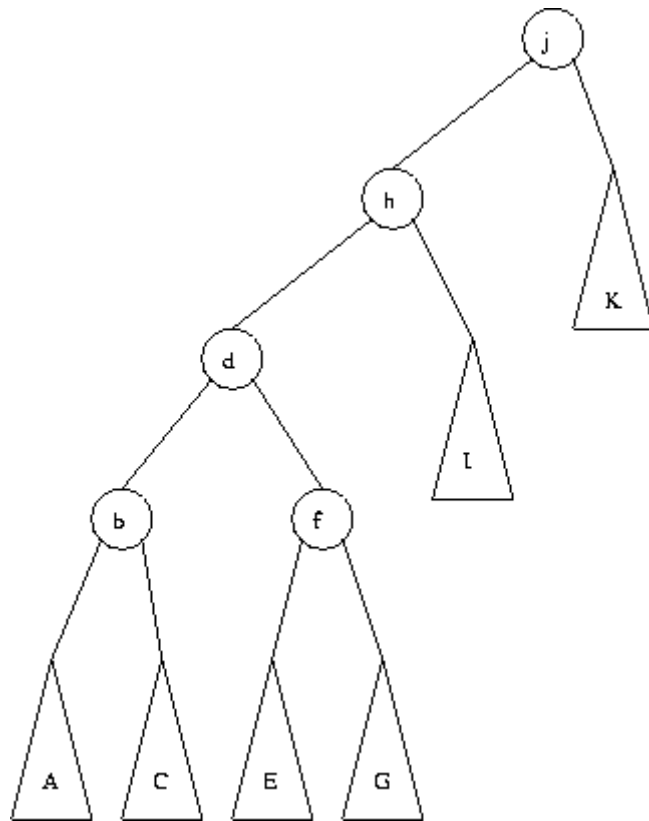
El efecto del splaying es no sólo de mover el nodo accesado a la raíz, sino que sube todos los nodos del camino desde la raíz hasta el nodo accesado aproximadamente a la mitad de su profundidad anterior, a costa que algunos pocos nodos bajen a lo más dos niveles en el árbol.

El siguiente ejemplo muestra como queda el splay tree luego de acceder al nodo  $d$ .

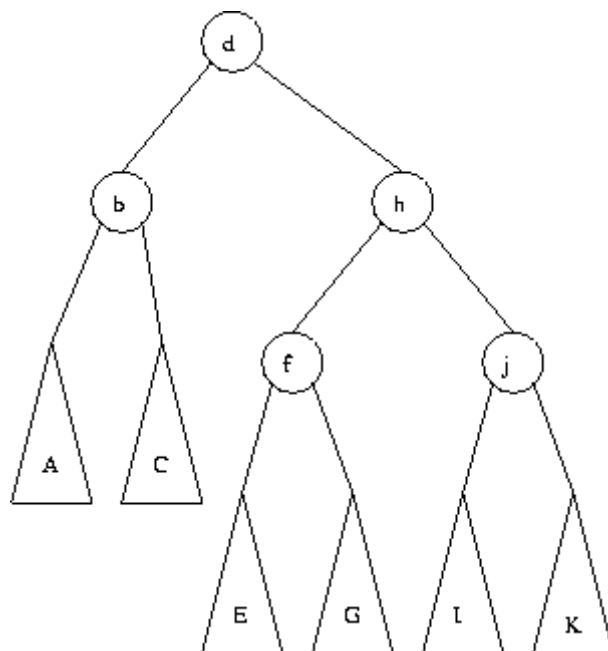


El primer paso es un *zig-zag* entre los nodos  $d$ ,  $b$  y  $f$ .





El último paso es un *zig-zig* entre los nodos  $d$ ,  $h$  y  $j$ .



Para borrar un elemento de un splay tree se puede proceder de la siguiente forma: se realiza una búsqueda del nodo a eliminar, lo cual lo deja en la raíz del árbol. Si ésta es eliminada, se obtienen dos subárboles  $S_{izq}$  y  $S_{der}$ . Se busca el elemento mayor en  $S_{izq}$ , con lo cual dicho elemento pasa a ser su nueva raíz, y como era el elemento mayor  $S_{izq}$  ya no tiene hijo derecho, por lo que se cuelga  $S_{der}$  como subárbol derecho de  $S_{izq}$ , con lo que termina la operación de eliminación.

El análisis de los splay trees es complejo, porque debe considerar la estructura cambiante del árbol en cada acceso realizado. Por otra parte, los splay trees son más fáciles de implementar que un AVL dado que no hay que verificar una condición de balance.

## Hashing

Suponga que desea almacenar  $n$  números enteros, sabiendo de antemano que dichos números se encuentran en un rango conocido  $0, \dots, k-1$ . Para resolver este problema, basta con crear un arreglo de valores booleanos de tamaño  $k$  y marcar con valor *true* los casilleros del arreglo cuyo índice sea igual al valor de los elementos a almacenar. Es fácil ver que con esta estructura de datos el costo de búsqueda, inserción y eliminación es  $O(1)$ .

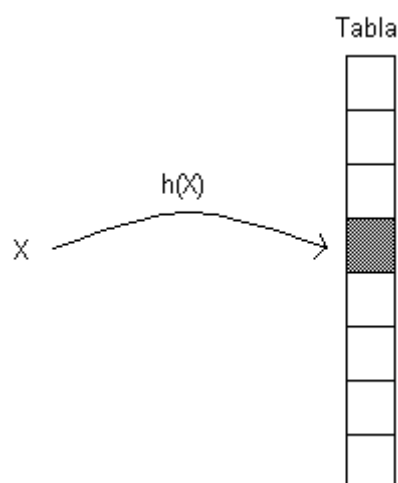
Elementos = {1, 4, 7}

|       |      |       |       |      |       |       |      |
|-------|------|-------|-------|------|-------|-------|------|
| 0     | 1    | 2     | 3     | 4    | 5     | 6     | 7    |
| false | true | false | false | true | false | false | true |

Este enfoque tiene dos grandes problemas:

- El valor de  $k$  puede ser muy grande, y por lo tanto no habría cupo en memoria para almacenar el arreglo. Piense, por ejemplo, en todos los posibles nombres de personas.
- Los datos a almacenar pueden ser pocos, con lo cual se estaría desperdiciando espacio de memoria.

Una manera de resolver estos problemas es usando una función  $h$ , denominada *función de hash*, que transforme un elemento  $X$ , perteneciente al universo de elementos posibles, en un valor  $h(X)$  en el rango  $[0, \dots, m-1]$ , con  $m \ll k$ . En este caso, se marca el casillero cuyo índice es  $h(X)$  para indicar que el elemento  $X$  pertenece al conjunto de elementos. Esta estructura de datos es conocida como *tabla de hashing*.



La función  $h$  debe ser de tipo *pseudoaleatorio* para distribuir las llaves uniformemente dentro de la tabla, es decir,  $Pr(h(X) = z) = 1/m$  para todo  $z$  en  $[0, \dots, m-1]$ . La llave  $X$  se puede interpretar como un número entero, y las funciones  $h(X)$  típicas son de la forma:

$$h(X) = (c \cdot X \bmod p) \bmod m$$

donde  $c$  es una constante,  $p$  es un número primo y  $m$  es el tamaño de la tabla de hashing. Distintos valores para estos parámetros producen distintas funciones de hash.

Problema de este nuevo enfoque: colisiones.

### La paradoja de los cumpleaños

¿Cuál es el número  $n$  mínimo de personas que es necesario reunir en una sala para que la probabilidad que dos de ella tengan su cumpleaños en el mismo día sea mayor que  $1/2$ ?

Sea  $d_n$  la probabilidad que no haya coincidencia en dos fechas de cumpleaños. Se tiene que:

$$d_n = \left(\frac{365}{365}\right) \left(\frac{364}{365}\right) \left(\frac{363}{365}\right) \cdots \left(\frac{365-n+1}{365}\right)$$

¿Cuál es el mínimo  $n$  tal que  $d_n < 1/2$ ?

Respuesta:  $n = 23 \Rightarrow d_n = 0.4927$ . Note que 23 es una "pequeña" fracción de 365.

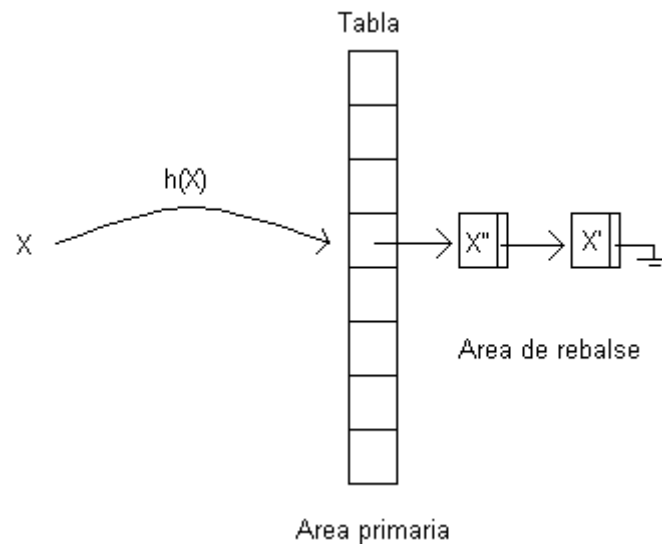
Esto quiere decir que con una pequeña fracción del conjunto de elementos es posible tener colisiones con alta probabilidad.

Para resolver el problema de las colisiones, existen dos grandes familias de métodos:

- Encadenamiento (usar estructuras dinámicas).
- Direccionamiento abierto (intentar con otra función de hash).

### **Encadenamiento**

La idea de este método es que todos los elementos que caen en el mismo casillero se enlacen en una lista, en la cual se realiza una búsqueda secuencial.



Se define para un conjunto con  $n$  elementos:

- $C_n$ : costo esperado de búsqueda exitosa.
- $C'_n$ : costo esperado de búsqueda infructuosa.
- $\alpha = \frac{n}{m}$  : factor de carga de la tabla.

$$C'_n = \frac{n}{m}, C_n = 1 + \frac{n-1}{2m} \Rightarrow C'_n = \alpha, C_n \approx 1 + \frac{\alpha}{2}$$

Esto implica que el costo esperado de búsqueda sólo depende del factor de carga  $\alpha$ , y no del tamaño de la tabla.

### Hashing con listas mezcladas

En este caso no se usa área de rebalse, sino que los elementos se almacenan en cualquier lugar libre del área primaria.

Ejemplo:  $h(X) = X \bmod 10$

|   |    |  |
|---|----|--|
| 0 |    |  |
| 1 |    |  |
| 2 |    |  |
| 3 |    |  |
| 4 |    |  |
| 5 | 25 |  |
| 6 |    |  |
| 7 | 48 |  |
| 8 | 15 |  |
| 9 | 39 |  |

Los costos esperados de búsqueda son:

$$C'_n = 1 + \frac{1}{4\alpha} (e^{2\alpha} - 1 - 2\alpha)$$

$$C_n = 1 + \frac{1}{8\alpha} (e^{2\alpha} - 1 - 2\alpha) + \frac{\alpha}{4}$$

### Eliminación en tablas de hashing con encadenamiento

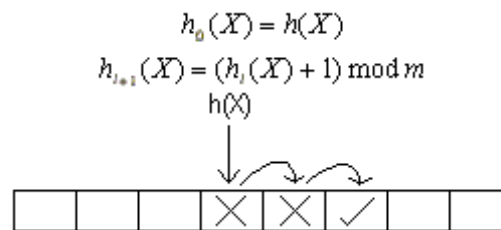
Con listas separadas el algoritmo es simple, basta con eliminar el elemento de la lista enlazada correspondiente. En el caso de las lista mezcladas, el algoritmo es más complejo (ejercicio).

### **Direccionamiento abierto**

En general, esto puede ser visto como una sucesión de funciones de hash  $\{h_0(X), h_1(X), \dots\}$ . Primero se intenta con  $tabla[h_0(X)]$ , si el casillero está ocupado se prueba con  $tabla[h_1(X)]$ , y así sucesivamente.

### Linear probing

Es el método más simple de direccionamiento abierto, en donde las funciones de hash se definen como:



Costo esperado de búsqueda:

$$C_n \approx \frac{1}{2} \left( 1 + \frac{1}{1-\alpha} \right)$$

$$C'_n \approx \frac{1}{2} \left( 1 + \frac{1}{(1-\alpha)^2} \right)$$

Para una tabla llena:

$$C_m = \Theta(\sqrt{m})$$

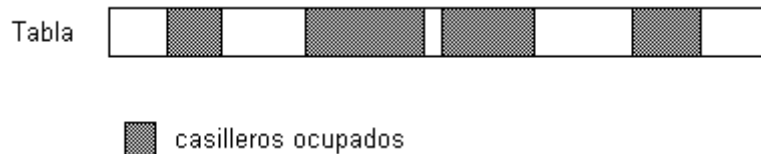
$$C'_{m-1} = \Theta(m)$$

Cuando la tabla de hashing está muy llena, este método resulta ser muy lento.

| $\alpha$ | $C_n$ | $C'_n$ |
|----------|-------|--------|
| .6       | 1.75  | 3.63   |
| .7       | 2.17  | 6.06   |
| .8       | 3     | 13     |

|      |        |            |
|------|--------|------------|
| .9   | 5.50   | 50.50      |
| .99  | 50.50  | 5,000.50   |
| .999 | 500.50 | 500,000.50 |

A medida que la tabla se va llenando, se observa que empiezan a aparecer *clusters* de casilleros ocupados consecutivos:



Si la función de hash distribuye los elementos uniformemente dentro de la tabla, la probabilidad que un cluster crezca es proporcional a su tamaño. Esto implica que una mala situación se vuelve peor cada vez con mayor probabilidad. Sin embargo, este no es todo el problema, puesto que lo mismo sucede en hashing con encadenamiento y no es tan malo. El verdadero problema ocurre cuando 2 clusters están separados solo por un casillero libre y ese casillero es ocupado por algún elemento: ambos clusters se unen en uno mucho más grande.

Otro problema que surge con linear probing es conocido como *clustering secundario*: si al realizar la búsqueda de dos elementos en la tabla se encuentran con el mismo casillero ocupado, entonces toda la búsqueda subsiguiente es la misma para ambos elementos.

Eliminación de elementos: no se puede eliminar un elemento y simplemente dejar su casillero vacío, puesto que las búsquedas terminarían en dicho casillero. Existen dos maneras para eliminar elementos de la tabla:

- Marcar el casillero como "eliminado", pero sin liberar el espacio. Esto produce que las búsquedas puedan ser lentas incluso si el factor de carga de la tabla es pequeño.
- Eliminar el elemento, liberar el casillero y mover elementos dentro de la tabla hasta que un casillero "verdaderamente" libre sea encontrado. Implementar esta operación es complejo y costoso.

### Hashing doble

En esta estrategia se usan dos funciones de hash: una función  $h(X) \in [0, \dots, m-1]$  conocida como *dirección inicial*, y una función  $s(X) \in [1, \dots, m-1]$  conocida como *paso*. Por lo tanto:

$$h_0(X) = h(X)$$

$$h_{i+1}(X) = (h_i(X) + s(X)) \bmod m$$

Elegir  $m$  primo asegura que se va a visitar toda la tabla antes que se empiecen a repetir los casilleros. Nota: solo basta que  $m$  y  $s(X)$  sean primos relativos (ejercicio: demostrarlo por contradicción).

El análisis de eficiencia de esta estrategia es muy complicado, y se estudian modelos idealizados: muestreo sin reemplazo (uniform probing) y muestreo con reemplazo (random probing), de los cuales se obtiene que los costos de búsqueda esperado son:

$$C_n \approx \frac{1}{\alpha} \ln \left( \frac{1}{1 - \alpha} \right)$$

$$C'_n \approx \frac{1}{1 - \alpha}$$

Para una tabla llena:

$$C_m = \Theta(\log m)$$

$$C'_{m-1} = \Theta(m)$$

Si bien las secuencias de casilleros obtenidas con hashing doble no son aleatorias, en la práctica su rendimiento es parecido a los valores obtenidos con los muestreos con y sin reemplazo.

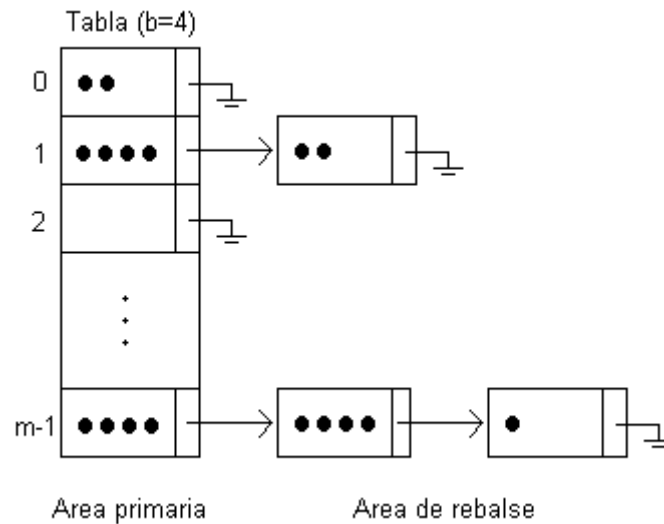
| $\alpha$ | $C_n$ | $C'_n$ |
|----------|-------|--------|
| .6       | 1.53  | 2.5    |
| .7       | 1.72  | 3.33   |
| .8       | 2.01  | 5      |
| .9       | 2.56  | 10     |
| .99      | 4.65  | 100    |
| .999     | 6.91  | 1,000  |

Existen heurísticas para resolver el problema de las colisiones en hashing con direccionamiento abierto, como por ejemplo *last-come-first-served* hashing (el elemento que se mueve de casillero no es el que se inserta sino el que ya lo ocupaba) y *Robin Hood* hashing (el elemento que se queda en el casillero es aquel que se encuentre más lejos de su posición original), que si bien mantienen el promedio de búsqueda con respecto al método original (*first-come-first-served*) disminuyen dramáticamente su varianza.

## Hashing en memoria secundaria

Cuando se almacena información en memoria secundaria (disco), la función de costo pasa a ser el número de accesos a disco. En hashing para memoria secundaria, la función de hash sirve para escoger un bloque (página) del disco, en donde cada bloque

contiene  $b$  elementos. El factor de carga pasa a ser  $\alpha = \frac{n}{mb}$ . Por ejemplo, utilizando hashing encadenado:

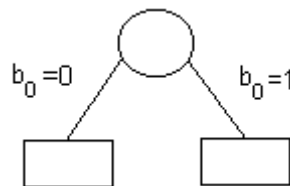


Este método es eficiente para un factor de carga pequeño, ya que con factor de carga alto la búsqueda toma tiempo  $O(n)$ . Para resolver esto puede ser necesario incrementar el tamaño de la tabla, y así reducir el factor de carga. En general esto implica reconstruir toda la tabla, pero existe un método que permite hacer crecer la tabla paulatinamente en el tiempo denominado *hashing extendible*.

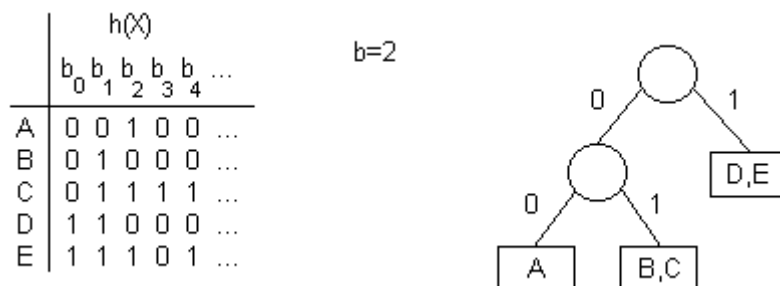
### Hashing extendible

Suponga que las páginas de disco son de tamaño  $b$  y una función de hash  $h(X) \geq 0$  (sin límite superior). Sea la descomposición en binario de la función de hash  $h(X) = (\dots b_2(X) b_1(X) b_0(X))_2$ .

Inicialmente, todas las llaves se encuentran en una única página. Cuando dicha página se rebalsa se divide en dos, usando  $b_0(X)$  para discriminar entre ambas páginas:

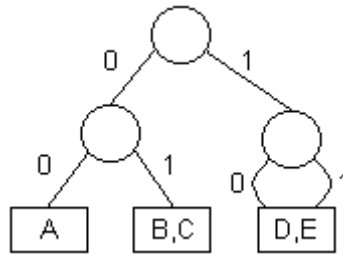


Cada vez que una página rebalsa, se usa el siguiente bit en la secuencia para dividirla en dos. Ejemplo:

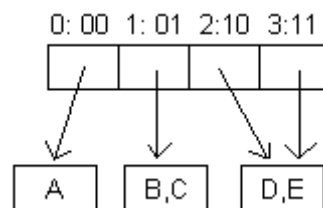




El índice (directorío) puede ser almacenado en un árbol, pero el hashing extensible utiliza una idea diferente. El árbol es extendido de manera que todos los caminos tengan el mismo largo:



A continuación, el árbol es implementado como un arreglo de referencias:



Cuando se inserta un elemento y existe un rebalse, si cae en la página  $(D,E)$  esta se divide y las referencias de los casilleros 2 y 3 apuntan a las nuevas páginas creadas. Si la página  $(B,C)$  es la que se divide, entonces el árbol crece un nivel, lo cual implica duplicar el tamaño del directorio. Sin embargo, esto no sucede muy a menudo.

El tamaño esperado del directorio es ligeramente superlineal:  $\Theta\left(m^{1+1/\delta}\right)$ .

# Compresión de datos

En esta sección veremos la aplicación de la teoría de árboles a la compresión de datos. Por compresión de datos entendemos cualquier algoritmo que reciba una cadena de datos de entrada y que sea capaz de generar una cadena de datos de salida cuya representación ocupa menos espacio de almacenamiento, y que permite -mediante un algoritmo de descompresión- recuperar total o parcialmente el mensaje recibido inicialmente. A nosotros nos interesa particularmente los algoritmos de compresión *sin pérdida*, es decir, aquellos algoritmos que permiten recuperar completamente la cadena de datos inicial.

## Codificación de mensajes

Supongamos que estamos codificando mensajes en binario con un alfabeto de tamaño  $n$ . Para esto se necesitan  $\lceil \log_2(n) \rceil$  bits por símbolo, si todos los códigos son de la misma longitud.

**Ejemplo:** para el alfabeto A, ..., Z de 26 letras se necesitan códigos de  $\lceil \log_2(26) \rceil = 5$  bits

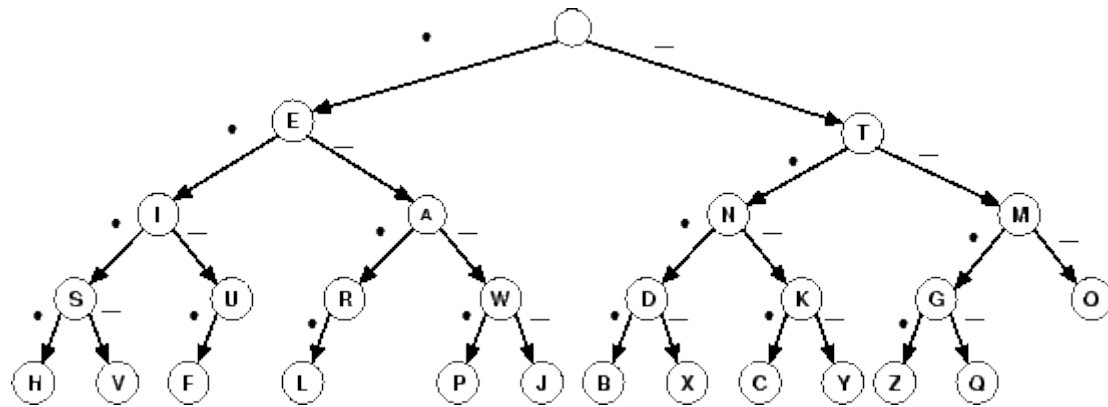
**Problema:** Es posible disminuir el número promedio de bits por símbolo?

**Solución:** Asignar códigos más cortos a los símbolos más frecuentes.

Un ejemplo claro de aplicación de este principio es el código morse:

A    . -  
B    - . . .  
:  
E    .  
:  
Z    - - . .

Se puede representar mediante un árbol binario:



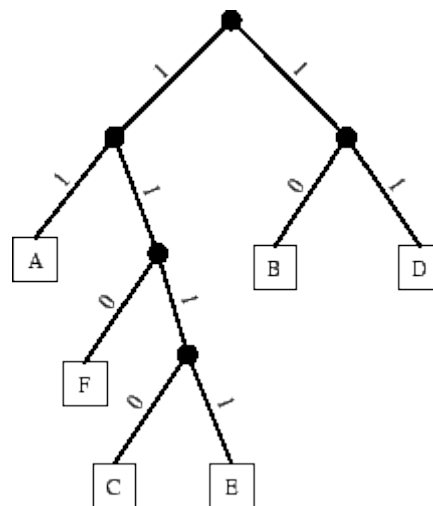
Podemos ver en el árbol que letras de mayor probabilidad de aparición (en idioma inglés) están más cerca de la raíz, y por lo tanto tienen una codificación más corta que letras de baja frecuencia.

**Problema:** este código no es auto-delimitante

Por ejemplo, SOS y IAMS tienen la misma codificación

Para eliminar estas ambigüedades, en morse se usa un tercer delimitador (espacio) para separar el código de cada letra. Se debe tener en cuenta que este problema se produce sólo cuando el código es de largo variable (como en morse), pues en otros códigos de largo fijo (por ejemplo el código ASCII, donde cada caracter se representa por 8 bits) es directo determinar cuales elementos componen cada caracter.

La condición que debe cumplir una codificación para no presentar ambigüedades, es que la codificación de ningún caracter sea prefijo de otra. Esto nos lleva a definir árboles que sólo tienen información en las hojas, como por ejemplo:



Como nuestro objetivo es obtener la secuencia codificada más corta posible, entonces tenemos que encontrar la codificación que en promedio use el menor largo promedio del código de cada letra.

**Problema:** Dado un alfabeto  $a_1, \dots, a_n$  tal que la probabilidad de que la letra  $a_i$  aparezca en un mensaje es  $p_i$ , encontrar un código libre de prefijos que minimice el largo promedio del código de una letra.

Supongamos que a la letra  $a_i$  se le asigna una codificación de largo  $t_i$ , entonces el largo esperado es:

$$\sum_i p_i t_i$$

es decir, el promedio ponderado de todas las letras por su probabilidad de aparición.

Ejemplo:

|   | probabilidad | código |
|---|--------------|--------|
| A | 0.30         | 00     |
| B | 0.25         | 10     |
| C | 0.08         | 0110   |
| D | 0.20         | 11     |
| E | 0.05         | 0111   |
| F | 0.12         | 010    |

$$\begin{aligned}
 \text{Costo esperado} &= 2(0,30 + 0,25 + 0,20) + 3(0,12) + 4(0,08 + 0,05) \\
 &= 2,075 + 0,36 + 0,52 \\
 &= 2,38 \text{ bits por simbolo}
 \end{aligned}$$

## Entropía de Shannon

Shannon define la entropía del alfabeto como:

$$-\sum_i p_i \log_2(p_i)$$

El teorema de Shannon dice que el número promedio de bits esperable para un conjunto de letras y probabilidades dadas se aproxima a la entropía del alfabeto. Podemos comprobar esto en nuestro ejemplo anterior donde la entropía de Shannon es:

$$\begin{aligned} \text{Entropía} &= -(0,30 \log_2(0,30) + 0,25 \log_2(0,25) + 0,08 \log_2(0,08) \\ &\quad + 0,2 \log_2(0,2) + 0,05 \log_2(0,05) + 0,12 \log_2(0,12)) \\ &= 0,521 + 0,5 + 0,2915 + 0,4643 + 0,2160 + 0,3670 \\ &\approx 2,36 \end{aligned}$$

que es bastante cercano al costo esperado de 2.38 que calculamos anteriormente.

A continuación describiremos algoritmos que nos permitan encontrar representaciones que minimicen el costo total.

## Algoritmo de Huffman

El algoritmo de Huffman permite construir un código libre de prefijos de costo esperado mínimo.

Inicialmente, comenzamos con  $n$  hojas desconectadas, cada una rotulada con una letra del alfabeto y con una probabilidad (ponderación o peso).

Consideremos este conjunto de hojas como un bosque. El algoritmo es:

```
while(nro de árboles del bosque > 1){
    - Encontrar los 2 árboles de peso mínimo y
      unirlos con una nueva raíz que se crea para esto.

    - Arbitrariamente, rotulamos las dos
      líneas como '0' y '1'

    - Darle a la nueva raíz un peso que es
      la suma de los pesos de sus subárboles.
}
```

### Ejemplo:

Si tenemos que las probabilidades de las letras en un mensaje son:

| letra | probabilidad |
|-------|--------------|
| a     | 0.121        |
| b     | 0.051        |
| c     | 0.137        |
| d     | 0.094        |
| e     | 0.274        |
| f     | 0.281        |
| g     | 0.042        |

Entonces la construcción del árbol de Huffman es (los números en **negrita** indican los árboles con menor peso):



Se puede ver que el costo esperado es de 2,53 bits por letra, mientras que una codificación de largo fijo (igual número de bits para cada símbolo) entrega un costo de 3 bits/letra.

El algoritmo de codificación de Huffman se basa en dos supuestos que le restan eficiencia:

1. supone que los caracteres son generados por una fuente aleatoria independiente, lo que en la práctica no es cierto. Por ejemplo, la probabilidad de encontrar una vocal después de una consonante es mucho mayor que la de encontrarla después de una vocal; después de una q es muy probable encontrar una u, etc
2. Debido a que la codificación se hace carácter a carácter, se pierde eficiencia al no considerar las secuencias de caracteres más probables que otras.

## Lempel Ziv

Una codificación que toma en cuenta los problemas de los supuestos enunciados anteriormente para la codificación de Huffman sería una donde no solo se consideraran caracteres uno a uno, sino que donde además se consideraran aquellas secuencias de alta probabilidad en el texto. Por ejemplo, en el texto:

*aaabbaabaa*

Obtendríamos un mayor grado de eficiencia si además de considerar caracteres como *a* y *b*, también considerásemos la secuencia *aa* al momento de codificar.

Una generalización de esta idea es el algoritmo de Lempel-Ziv. Este algoritmo consiste en separar la secuencia de caracteres de entrada en bloques o secuencias de caracteres de distintos largos, manteniendo un diccionario de bloques ya vistos. Aplicando el algoritmo de Huffman para estos bloques y sus probabilidades, se puede sacar provecho de las secuencias que se repitan con más probabilidad en el texto. El algoritmo de codificación es el siguiente:

- 1.- Inicializar el diccionario con todos los bloques de largo 1
- 2.- Seleccionar el prefijo más largo del mensaje que calce con alguna secuencia **W** del diccionario y eliminar **W** del mensaje
- 3.- Codificar **W** con su índice en el diccionario
- 4.- Agregar **W** seguido del primer símbolo del próximo bloque al diccionario.
- 5.- Repetir desde el paso 2.



Ejemplo:

Si el mensaje es

*abbaabbaababbaaaabaabba*

la codificación y los bloques agregados al diccionario serían (donde los bloques reconocidos son mostrados entre paréntesis y la secuencia agregada al diccionario en cada etapa es mostrada como un subíndice):

$(a)_{ab}(b)_{bb}(b)_{ba}(a)_{aa}(ab)_{abb}(ba)_{baa}(ab)_{aba}(abb)_{abba}(aa)_{aaa}(aa)_{aab}(baa)_{baab}(bb)_{bba}(a)$

Teóricamente, el diccionario puede crecer indefinidamente, pero en la práctica se opta por tener un diccionario de tamaño limitado. Cuando se llega al límite del diccionario, no se agregan más bloques.

Lempel-Ziv es una de las alternativas a Huffman. Existen varias otras derivadas de estas dos primeras, como LZW (Lempel-Ziv-Welch), que es usado en programas de compresión como el *compress* de UNIX.

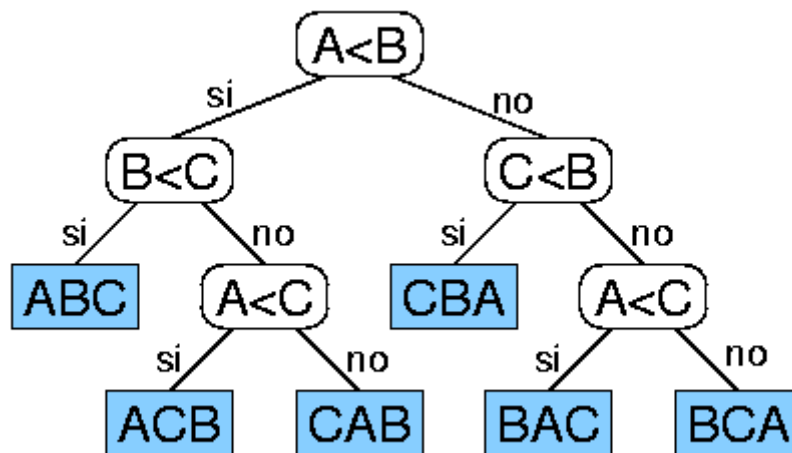
# Ordenación

1. [Cota inferior](#)
2. [Quicksort](#)
3. [Heapsort](#)
4. [Bucketsort](#)
5. [Mergesort y Ordenamiento Externo](#)

El problema de ordenar un conjunto de datos (por ejemplo, en orden ascendente) tiene gran importancia tanto teórica como práctica. En esta sección veremos principalmente algoritmos que ordenan mediante comparaciones entre llaves, para los cuales se puede demostrar una cota inferior que coincide con la cota superior provista por varios algoritmos. También veremos un algoritmo de otro tipo, que al no hacer comparaciones, no está sujeto a esa cota inferior.

## Cota inferior

Supongamos que deseamos ordenar tres datos A, B y C. La siguiente figura muestra un árbol de decisión posible para resolver este problema. Los nodos internos del árbol representan comparaciones y los nodos externos representan salidas emitidas por el programa.



Como se vio en el capítulo de búsqueda, todo árbol de decisión con  $H$  hojas tiene al menos altura  $\log_2 H$ , y la altura del árbol de decisión es igual al número de comparaciones que se efectúan en el peor caso.

En un árbol de decisión para ordenar  $n$  datos se tiene que  $H=n!$ , y por lo tanto se tiene que todo algoritmo que ordene  $n$  datos mediante comparaciones entre llaves debe hacer al menos  $\log_2 n!$  comparaciones en el peor caso.

Usando la aproximación de Stirling, se puede demostrar que  $\log_2 n! = n \log_2 n + O(n)$ , por lo cual la cota inferior es de  $O(n \log n)$ .

Si suponemos que todas las posibles permutaciones resultantes son equiprobables, es posible demostrar que el número *promedio* de comparaciones que cualquier algoritmo debe hacer es también de  $O(n \log n)$ .

## Quicksort

Este método fue inventado por C.A.R. Hoare a comienzos de los '60s, y sigue siendo el método más eficiente para uso general.

Quicksort es un ejemplo clásico de la aplicación del principio de *dividir para reinar*. Su estructura es la siguiente:

- Primero se elige un elemento al azar, que se denomina el *pivote*.
- El arreglo a ordenar se reordena dejando a la izquierda a los elementos menores que el pivote, el pivote al medio, y a la derecha los elementos mayores que el pivote:



- Luego cada sub-arreglo se ordena recursivamente.

La recursividad termina, en principio, cuando se llega a sub-arreglos de tamaño cero o uno, los cuales trivialmente ya están ordenados. En la práctica veremos que es preferible detener la recursividad antes de eso, para no desperdiciar tiempo ordenando recursivamente arreglos pequeños, los cuales pueden ordenarse más eficientemente usando Ordenación por Inserción, por ejemplo.

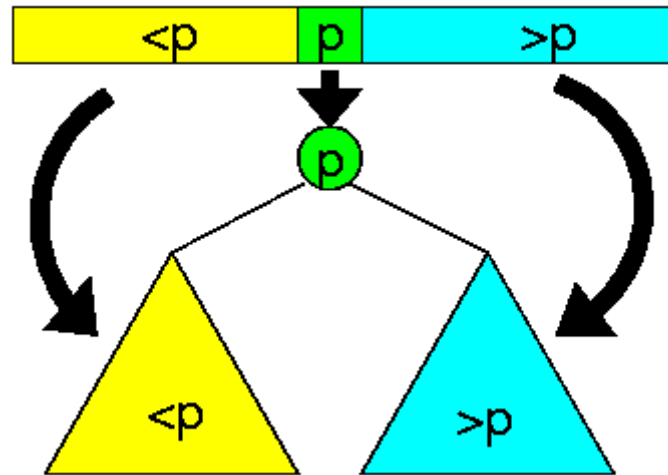
### Costo promedio

Si suponemos, como una primera aproximación, que el pivote siempre resulta ser la mediana del conjunto, entonces el costo de ordenar está dado (aproximadamente) por la ecuación de recurrencia

$$T(n) = n + 2 T(n/2)$$

Esto tiene solución  $T(n) = n \log_2 n$  y es, en realidad, el *mejor* caso de Quicksort.

Para analizar el tiempo promedio que demora la ordenación mediante Quicksort, observemos que el funcionamiento de Quicksort puede graficarse mediante un *árbol de partición*:



Por la forma en que se construye, es fácil ver que el árbol de partición es un *árbol de búsqueda binaria*, y como el pivote es escogido al azar, entonces la raíz de cada subárbol puede ser cualquiera de los elementos del conjunto en forma equiprobable. En consecuencia, los árboles de partición y los árboles de búsqueda binaria tienen exactamente la misma distribución.

En el proceso de partición, cada elemento de los subárboles ha sido comparado contra la raíz (el pivote). Al terminar el proceso, cada elemento ha sido comparado contra todos sus ancestros. Si sumamos todas estas comparaciones, el resultado total es igual al *largo de caminos internos*.

Usando todas estas correspondencias, tenemos que, usando los resultados ya conocidos para árboles, el número promedio de comparaciones que realiza Quicksort es de:

$$1.38 n \log_2 n + O(n)$$

Por lo tanto, Quicksort es del mismo orden que la cota inferior (en el caso esperado).

## Peor caso

El peor caso de Quicksort se produce cuando el pivote resulta ser siempre el mínimo o el máximo del conjunto. En este caso la ecuación de recurrencia es

$$T(n) = n - 1 + T(n-1)$$

lo que tiene solución  $T(n) = O(n^2)$ . Desde el punto de vista del árbol de partición, esto corresponde a un árbol en "zig-zag".

Si bien este peor caso es extremadamente improbable si el pivote se escoge al azar, algunas implementaciones de Quicksort toman como pivote al primer elemento del arreglo (suponiendo que, al venir el arreglo al azar, entonces el primer elemento es tan aleatorio como cualquier otro). El problema es que si el conjunto viene en realidad *ordenado*, entonces caemos justo en el peor caso cuadrático.

Lo anterior refuerza la importancia de que el pivote se escoja al azar. Esto no aumenta significativamente el costo total, porque el número total de elecciones de pivote es  $O(n)$ .

## Mejoras a Quicksort

Quicksort puede ser optimizado de varias maneras, pero hay que ser muy cuidadoso con estas mejoras, porque es fácil que terminen empeorando el desempeño del algoritmo.

En primer lugar, es desaconsejable hacer cosas que aumenten la cantidad de trabajo que se hace dentro del "loop" de partición, porque este es el lugar en donde se concentra el costo  $O(n \log n)$ .

Algunas de las mejoras que han dado buen resultado son las siguientes:

### *Quicksort con "mediana de 3"*

En esta variante, el pivote no se escoge como un elemento tomado al azar, sino que primero se extrae una muestra de 3 elementos, y entre ellos se escoge a la mediana de esa muestra como pivote.

Si la muestra se escoge tomando al primer elemento del arreglo, al del medio y al último, entonces lo que era el peor caso (arreglo ordenado) se transforma de inmediato en mejor caso.

De todas formas, es aconsejable que la muestra se escoja al azar, y en ese caso el análisis muestra que el costo esperado para ordenar  $n$  elementos es

$$(12/7) n \ln n = 1.19 n \log_2 n$$

Esta reducción en el costo se debe a que el pivote es ahora una mejor aproximación a la mediana. De hecho, si en lugar de escoger una muestra de tamaño 3, lo hiciéramos con tamaños como 7, 9, etc., se lograría una reducción aún mayor, acercándonos cada vez más al óptimo, pero con rendimientos rápidamente decrecientes.

### *Uso de Ordenación por Inserción para ordenar sub-arreglos pequeños*

Tal como se dijo antes, no es eficiente ordenar recursivamente sub-arreglos demasiado pequeños.

En lugar de esto, se puede establecer un tamaño mínimo  $M$ , de modo que los sub-arreglos de tamaño menor que esto se ordenan por inserción en lugar de por Quicksort.

Claramente debe haber un valor óptimo para  $M$ , porque si creciera indefinidamente se llegaría a un algoritmo cuadrático. Esto se puede analizar, y el óptimo es cercano a 10.

Como método de implementación, al detectarse un sub-arreglo de tamaño menor que  $M$ , se lo puede dejar simplemente *sin ordenar*, retornando de inmediato de la recursividad. Al final del proceso, se tiene un arreglo cuyos pivotes están en orden creciente, y encierran entre ellos a bloques de elementos desordenados, pero que ya están en el

grupo correcto. Para completar la ordenación, entonces, basta con hacer una sola gran pasada de Ordenación por Inserción, la cual ahora no tiene costo  $O(n^2)$ , sino  $O(nM)$ , porque ningún elemento está a distancia mayor que  $M$  de su ubicación definitiva.

### ***Ordenar recursivamente sólo el sub-arreglo más pequeño***

Un problema potencial con Quicksort es la profundidad que puede llegar a tener el arreglo de recursividad. En el peor caso, ésta puede llegar a ser  $O(n)$ .

Para evitar esto, vemos primero cómo se puede programar Quicksort en forma no recursiva, usando un stack. El esquema del algoritmo sería el siguiente (en pseudo-Java):

```
void Quicksort(Object a[])
{
    Pila S = new Pila();
    S.apilar(1,N); // límites iniciales del arreglo
    while(!S.estaVacía())
    {
        (i,j) = S.desapilar(); // sacar límites
        if(j-i>0) // al menos dos elementos para ordenar
        {
            p = particionar(a,i,j); // pivote queda en a[p]
            S.apilar(i,p-1);
            S.apilar(p+1,j);
        }
    }
}
```

Con este enfoque se corre el riesgo de que la pila llegue a tener profundidad  $O(n)$ . Para evitar esto, podemos colocar en la pila sólo los límites del sub-arreglo más pequeño, dejando el más grande para ordenarlo de inmediato, sin pasar por la pila:

```
void Quicksort(Object a[])
{
    Pila S = new Pila();
    S.apilar(1,N); // límites iniciales del arreglo
    while(!S.estaVacía())
    {
        (i,j) = S.desapilar(); // sacar límites
        while(j-i>0) // al menos dos elementos para ordenar
        {
            p = particionar(a,i,j); // pivote queda en a[p]
            if(p-i>j-p) // mitad izquierda es mayor
            {
                S.apilar(p+1,j);
                j=p-1;
            }
            else
            {
                S.apilar(i,p-1);
                i=p+1;
            }
        }
    }
}
```

Con este enfoque, cada intervalo apilado es a lo más de la mitad del tamaño del arreglo, de modo que si llamamos  $S(n)$  a la profundidad de la pila, tenemos:

$$S(n) \leq 1 + S(n/2)$$

lo cual tiene solución  $\log_2 n$ , de modo que la profundidad de la pila nunca es más que logarítmica.

## Un algoritmo de selección basado en Quicksort

Es posible modificar el algoritmo de Quicksort para seleccionar el  $k$ -ésimo elemento de un arreglo. Básicamente, la idea es ejecutar Quicksort, pero en lugar de ordenar las dos mitades, hacerlo solo con aquella mitad en donde se encontraría el elemento buscado.

Suponemos que los elementos están en  $a[1], \dots, a[n]$  y que  $k$  está entre 1 y  $n$ . Cuando el algoritmo termina, el  $k$ -ésimo elemento se encuentra en  $a[k]$ . El resultado es el siguiente algoritmo, conocido como Quickselect, el cual se llama inicialmente como `Quickselect(a, k, 1, N)`.

```
void Quickselect(Object a[], int k, int i, int j)
{
    if(j-i>0) // aún quedan al menos 2 elementos
    {
        p = particionar(a,i,j);
        if(p==k) // ¡bingo!
            return;
        if(k<p) // seguimos buscando a la izquierda
            Quickselect(a,k,i,p-1);
        else
            Quickselect(a,k,p+1,j);
    }
}
```

Dado que en realidad se hace sólo una llamada recursiva y que ésta es del tipo "tail recursion", es fácil transformar esto en un algoritmo iterativo (hacerlo como ejercicio).

El análisis de Quickselect es difícil, pero se puede demostrar que el costo esperado es  $O(n)$ . Sin embargo, el peor caso es  $O(n^2)$ .

## Heapsort

A partir de cualquier implementación de una [cola de prioridad](#) es posible obtener un algoritmo de ordenación. El esquema del algoritmo es:

- Comenzar con una cola de prioridad vacía.
- *Fase de construcción de la cola de prioridad:*  
Traspasar todos los elementos del conjunto que se va a ordenar a la cola de prioridad, mediante  $n$  inserciones.
- *Fase de ordenación:*  
Sucesivamente extraer el máximo  $n$  veces. Los elementos van apareciendo en orden decreciente y se van almacenando en el conjunto de salida.

Si aplicamos esta idea a las dos implementaciones simples de colas de prioridad, utilizando lista enlazada ordenada y lista enlazada desordenada, se obtienen los algoritmos de ordenación por Inserción y por Selección, respectivamente. Ambos son algoritmos cuadráticos, pero es posible que una mejor implementación lleve a un algoritmo más rápido. En el capítulo de Tipos de Datos Abstractos se vió que una forma de obtener una implementación eficiente de colas de prioridad es utilizando una estructura de datos llamada *heap*.

## **Implementación de Heapsort**

Al aplicar heaps en la implementación de cola de prioridad para construir un algoritmo de ordenación, se obtiene un algoritmo llamado Heapsort, para el cual resulta que tanto la fase de construcción de la cola de prioridad, como la fase de ordenación, tienen ambas costo  $O(n \log n)$ , de modo que el algoritmo completo tiene ese mismo costo.

Por lo tanto, Heapsort tiene un orden de magnitud que coincide con la cota inferior, esto es, es óptimo incluso en el peor caso. Nótese que esto no era así para Quicksort, el cual era óptimo en promedio, pero no en el peor caso.

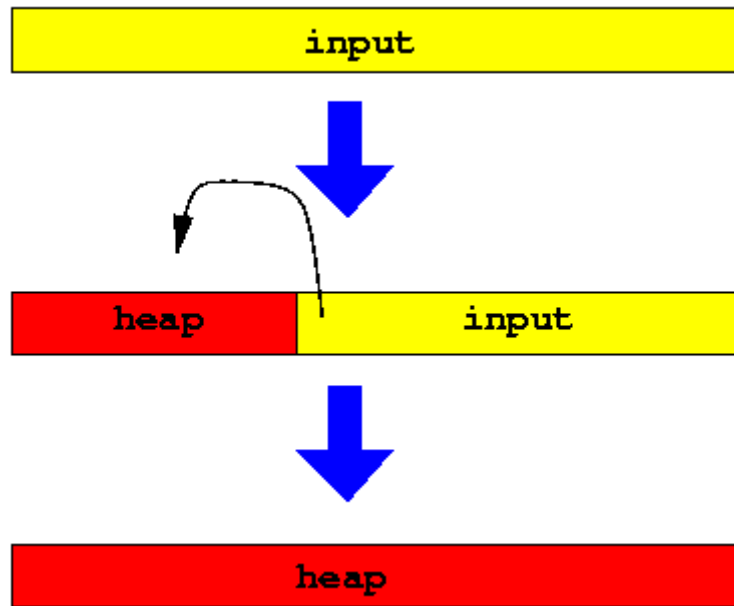
De acuerdo a la descripción de esta familia de algoritmos, daría la impresión de que en la fase de construcción del heap se requeriría un arreglo aparte para el heap, distinto del arreglo de entrada. De la misma manera, se requeriría un arreglo de salida aparte, distinto del heap, para recibir los elementos a medida que van siendo extraídos en la fase de ordenación.

En la práctica, esto no es necesario y basta con un sólo arreglo: todas las operaciones pueden efectuarse directamente sobre el arreglo de entrada.

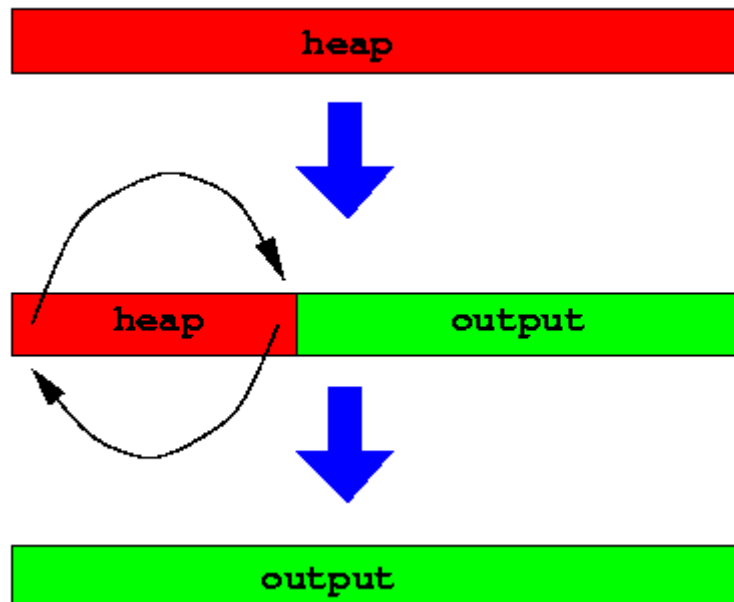
En primer lugar, en cualquier momento de la ejecución del algoritmo, los elementos se encuentran particionados entre aquellos que están ya o aún formando parte del heap, y aquellos que se encuentran aún en el conjunto de entrada, o ya se encuentran en el conjunto de salida, según sea la fase. Como ningún elemento puede estar en más de un conjunto a la vez, es claro que, en todo momento, en total nunca se necesita más de  $n$  casilleros de memoria, si la implementación se realiza bien.

En el caso de Heapsort, durante la fase de construcción del heap, podemos utilizar las celdas de la izquierda del arreglo para ir "armando" el heap. Las celdas necesarias para ello se las vamos "quitando" al conjunto de entrada, el cual va perdiendo elementos a medida que se insertan en el heap. Al concluir esta fase, todos los elementos han sido insertados, y el arreglo completo es un solo gran heap.





En la fase de ordenación, se van extrayendo elementos del heap, con lo cual este se contrae de tamaño y deja espacio libre al final, el cual puede ser justamente ocupado para ir almacenando los elementos a medida que van saliendo del heap (recordemos que van apareciendo en orden decreciente).



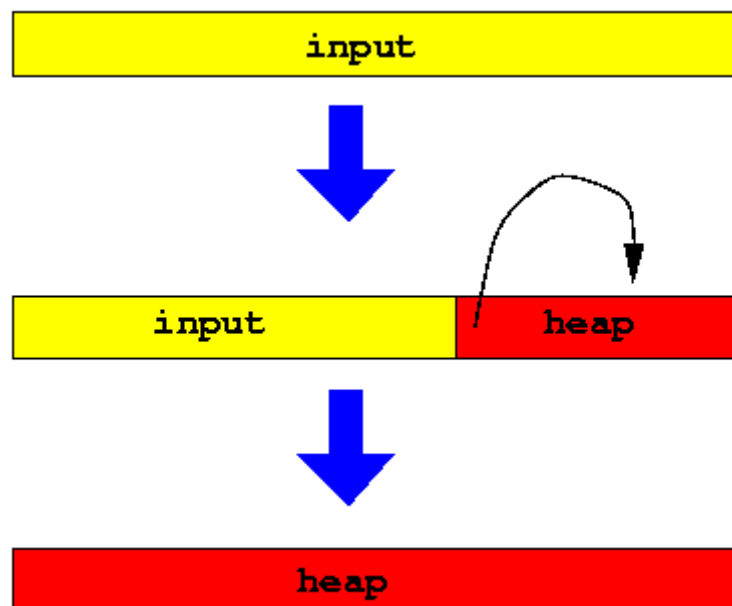
### Optimización de la fase de construcción del heap

Como se ha señalado anteriormente, tanto la fase de construcción del heap como la de ordenación demoran tiempo  $O(n \log n)$ . Esto es el mínimo posible (en orden de magnitud), de modo que no es posible mejorarlo significativamente.

Sin embargo, es posible modificar la implementación de la fase de construcción del heap para que sea mucho más eficiente.

La idea es invertir el orden de las "mitades" del arreglo, haciendo que el "input" esté a la izquierda y el "heap" a la derecha.

En realidad, si el "heap" está a la derecha, entonces no es realmente un heap, porque no es un árbol completo (le falta la parte superior), pero sólo nos interesa que en ese sector del arreglo se cumplan las relaciones de orden entre  $a[k]$  y  $\{a[2*k], a[2*k+1]\}$ . En cada iteración, se toma el último elemento del "input" y se le "hunde" dentro del heap de acuerdo a su nivel de prioridad.



Al concluir, se llega igualmente a un heap completo, pero el proceso es significativamente más rápido.

La razón es que, al ser "hundido", un elemento paga un costo proporcional a su distancia al fondo del árbol. Dada las características de un árbol, la gran mayoría de los elementos están al fondo o muy cerca de él, por lo cual pagan un costo muy bajo. En un análisis aproximado, la mitad de los elementos pagan 0 (ya están al fondo), la cuarta parte paga 1, la octava parte paga 2, etc. Sumando todo esto, tenemos que el costo total está acotado por

$$n \left( \sum_{i=1}^{\infty} \frac{i-1}{2^i} \right)$$

lo cual es igual a  $n$ .

## Bucketsort

Los métodos anteriores operan mediante comparaciones de llaves, y están sujetos, por lo tanto, a la cota inferior  $O(n \log n)$ . Veremos a continuación un método que opera de una manera distinta, y logra ordenar el conjunto en tiempo lineal.

Supongamos que queremos ordenar  $n$  números, cada uno de ellos compuesto de  $k$  dígitos decimales. El siguiente es un ejemplo con  $n=10$ ,  $k=5$ .

73895  
93754  
82149  
99046  
04853  
94171  
54963  
70471  
80564  
66496

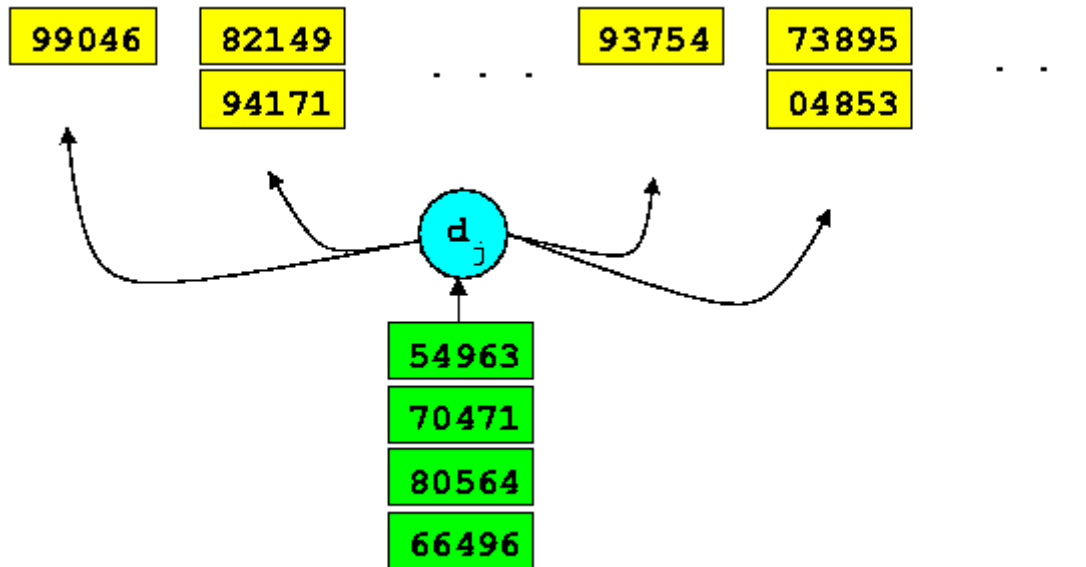
Imaginando que estos dígitos forman parte de una matriz, podemos decir que  $a[i, j]$  es el  $j$ -ésimo del  $i$ -ésimo elemento del conjunto.

Es fácil, en una pasada, ordenar el conjunto si la llave de ordenación es *un* solo dígito, por ejemplo el tercero de izquierda a derecha:

99**0**46  
82**1**49  
94**1**71  
70**4**71  
66**4**96  
80**5**64  
93**7**54  
73**8**95  
04**8**53  
54**9**63

Llamemos  $j$  a la posición del dígito mediante el cual se ordena. La ordenación se puede hacer utilizando una cola de entrada, que contiene al conjunto a ordenar, y un arreglo de 10 colas de salida, subindicadas de 0 a 9. Los elementos se van sacando de la cola de entrada y se van encolando en la cola de salida  $Q[d_j]$ , donde  $d_j$  es el  $j$ -ésimo dígito del elemento que se está transfiriendo.

Al terminar este proceso, los elementos están separados por dígito. Para completar la ordenación, basta concatenar las  $k$  colas de salida y formar nuevamente una sola cola con todos los elementos.



Este proceso se hace en una pasada sobre los datos, y toma tiempo  $O(n)$ .

Para ordenar el conjunto por las llaves completas, repetimos el proceso dígito por dígito, en cada pasada separando los elementos según el valor del dígito respectivo, luego recolectándolos para formar una sola cola, y realimentando el proceso con esos mismos datos. El conjunto completo queda finalmente ordenado si los dígitos se van tomando *de derecha a izquierda* (¡esto no es obvio!).

Como hay que realizar  $k$  pasadas y cada una de ellas toma tiempo  $O(n)$ , el tiempo total es  $O(k \cdot n)$ , que es el tamaño del archivo de entrada (en bytes). Por lo tanto, la ordenación toma tiempo lineal en el tamaño de los datos.

El proceso anterior se puede generalizar para cualquier alfabeto, no sólo dígitos (por ejemplo, ASCII). Esto aumenta el número de colas de salida, pero no cambia sustancialmente el tiempo que demora el programa.

### Archivos con records de largo variable

Si las líneas a ordenar no son todas del mismo largo, es posible alargarlas hasta completar el largo máximo, con lo cual el algoritmo anterior es aplicable. Pero si hay algunas pocas líneas desproporcionadamente largas y otras muy cortas, se puede perder mucha eficiencia.

Es posible, aunque no lo vamos a ver aquí, generalizar este algoritmo para ordenar líneas de largo variable sin necesidad de alargarlas. El resultado es que la ordenación se realiza en tiempo proporcional al tamaño del archivo.

## Mergesort y Ordenamiento Externo

Si tenemos dos archivos que ya están ordenados, podemos mezclarlos para formar un solo archivo ordenado en tiempo proporcional a la suma de los tamaños de los dos archivos.

Esto se hace leyendo el primer elemento de cada archivo, copiando hacia la salida al menor de los dos, y avanzando al siguiente elemento en el archivo respectivo. Cuando uno de los dos archivos se termina, todos los elementos restantes del otro se copian hacia la salida. Este proceso se denomina "mezcla", o bien "merge", por su nombre en inglés.

Como cada elemento se copia sólo una vez, y con cada comparación se copia algún elemento, es evidente que el costo de mezclar los dos archivos es lineal.

Si bien es posible realizar el proceso de mezcla de dos arreglos contiguos *in situ*, el algoritmo es muy complicado y no resulta práctico. Por esta razón, el proceso se implementa generalmente copiando de un archivo a otro.

Usando esta idea en forma reiterada, es posible ordenar un conjunto. Una forma de ver esto es recursivamente, usando "dividir para reinar". El siguiente pseudo-código ilustra esta idea:

```
mergesort(S) # retorna el conjunto S ordenado
{
    if(S es vacío o tiene sólo 1 elemento)
        return(S);
    else
    {
        Dividir S en dos mitades A y B;
        A'=mergesort(A);
        B'=mergesort(B);
        return(merge(A',B')) ;
    }
}
```

El tiempo total está dado aproximadamente por la ecuación de recurrencia

$$T(n) = 2 T(n/2) + n$$

la cual tiene solución  $O(n \log n)$ , de modo que el algoritmo resulta ser óptimo.

Esto mismo se puede implementar en forma no recursiva, agrupando los elementos de a dos y mezclándolos para formar pares ordenados. Luego mezclamos pares para formar cuádruplas ordenadas, y así sucesivamente hasta mezclar las últimas dos mitades y formar el conjunto completo ordenado. Como cada "ronda" tiene costo lineal y se realizan  $\log n$  rondas, el costo total es  $O(n \log n)$ .

La idea de Mergesort es la base de la mayoría de los métodos de ordenamiento externo, esto es, métodos que ordenan conjuntos almacenados en archivos muy grandes, en donde no es posible copiar todo el contenido del archivo a memoria para aplicar alguno de los métodos estudiados anteriormente.

En las implementaciones prácticas de estos métodos, se utiliza el enfoque no recursivo, optimizado usando las siguientes ideas:

- No se comienza con elementos individuales para formar pares ordenados, sino que se generan archivos ordenados lo más grandes posibles. Para esto, el archivo de entrada se va leyendo por trozos a memoria y se ordena mediante Quicksort, por ejemplo.
- En lugar de mezclar sólo dos archivos se hace una mezcla múltiple (con  $k$  archivos de entrada. Como en cada iteración hay  $k$  candidatos a ser el siguiente elemento en salir, y siempre hay que extraer al mínimo de ellos y sustituirlo en la lista de candidatos por su sucesor, la estructura de datos apropiada para ello es un heap.

En caso que no baste con una pasada de mezcla múltiple para ordenar todo el archivo, el proceso se repite las veces que sea necesario.

Al igual que en los casos anteriores, el costo total es  $O(n \log n)$ .

# Búsqueda en texto

1. [Algoritmo de fuerza bruta.](#)
2. [Algoritmo Knuth-Morris-Pratt \(KMP\).](#)
3. [Algoritmo Boyer-Moore.](#)
  - [Boyer-Moore-Horspool \(BMH\).](#)
  - [Boyer-Moore-Sunday \(BMS\).](#)

La búsqueda de patrones en un texto es un problema muy importante en la práctica. Sus aplicaciones en computación son variadas, como por ejemplo la búsqueda de una palabra en un archivo de texto o problemas relacionados con biología computacional, en donde se requiere buscar patrones dentro de una secuencia de ADN, la cual puede ser modelada como una secuencia de caracteres (el problema es más complejo que lo descrito, puesto que se requiere buscar patrones en donde ocurren alteraciones con cierta probabilidad, esto es, la búsqueda no es exacta).

En este capítulo se considerará el problema de buscar la ocurrencia de un patrón dentro de un texto. Se utilizarán las siguientes convenciones:

- $n$  denotará el largo del texto en donde se buscará el patrón, es decir,  $texto = a_1 a_2 \dots a_n$ .
- $m$  denotará el largo del patrón a buscar, es decir,  $patrón = b_1 b_2 \dots b_m$ .

Por ejemplo:

- Texto = "análisis de algoritmos".
- Patrón = "algo".

## Algoritmo de fuerza bruta

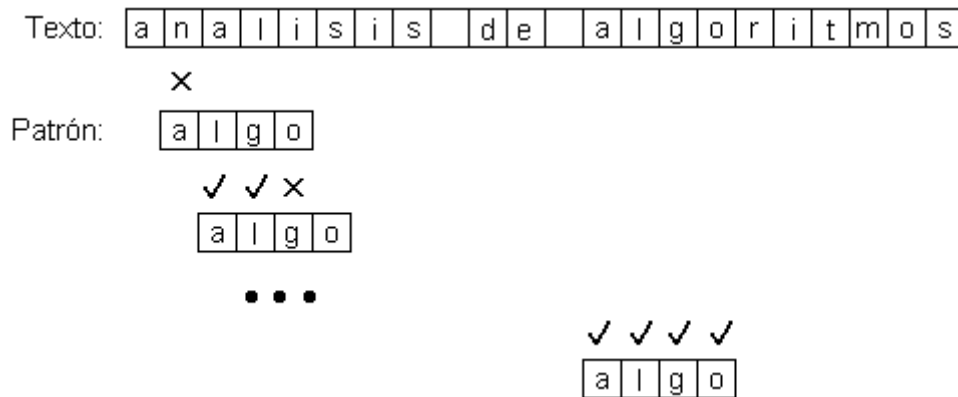
Se alinea la primera posición del patrón con la primera posición del texto, y se comparan los caracteres uno a uno hasta que se acabe el patrón, esto es, se encontró una ocurrencia del patrón en el texto, o hasta que se encuentre una discrepancia.

|        |   |   |   |   |   |   |   |   |  |   |   |  |   |   |   |   |   |   |   |   |   |   |
|--------|---|---|---|---|---|---|---|---|--|---|---|--|---|---|---|---|---|---|---|---|---|---|
| Texto: | a | n | a | l | i | s | i | s |  | d | e |  | a | l | g | o | r | i | t | m | o | s |
|--------|---|---|---|---|---|---|---|---|--|---|---|--|---|---|---|---|---|---|---|---|---|---|

✓ ×

|         |   |   |   |   |
|---------|---|---|---|---|
| Patrón: | a | l | g | o |
|---------|---|---|---|---|

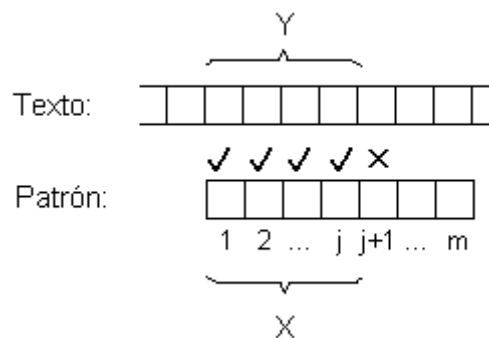
Si se detiene la búsqueda por una discrepancia, se desliza el patrón en una posición hacia la derecha y se intenta calzar el patrón nuevamente.



En el peor caso este algoritmo realiza  $O(mn)$  comparaciones de caracteres.

## Algoritmo Knuth-Morris-Pratt (KMP)

Suponga que se está comparando el patrón y el texto en una posición dada, cuando se encuentra una discrepancia.



Sea  $X$  la parte del patrón que calza con el texto, e  $Y$  la correspondiente parte del texto, y suponga que el largo de  $X$  es  $j$ . El algoritmo de fuerza bruta mueve el patrón una posición hacia la derecha, sin embargo, esto puede o no puede ser lo correcto en el sentido que los primeros  $j-1$  caracteres de  $X$  pueden o no pueden calzar los últimos  $j-1$  caracteres de  $Y$ .

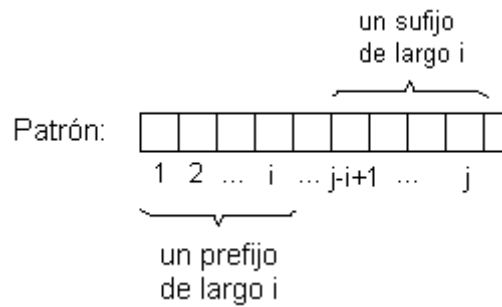
La observación clave que realiza el algoritmo Knuth-Morris-Pratt (en adelante KMP) es que  $X$  es igual a  $Y$ , por lo que la pregunta planteada en el párrafo anterior puede ser respondida mirando solamente el patrón de búsqueda, lo cual permite precalcular la respuesta y almacenarla en una tabla.

Por lo tanto, si deslizar el patrón en una posición no funciona, se puede intentar deslizarlo en 2, 3, ..., hasta  $j$  posiciones.

Se define la *función de fracaso* (failure function) del patrón como:

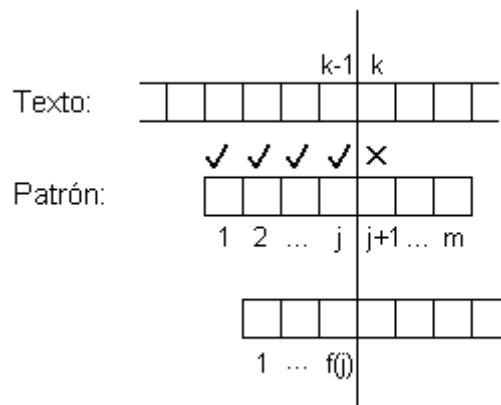
$$f(j) = \max\{i < j \mid b_1 \dots b_i = b_{j-i+1} \dots b_j\}$$





Intuitivamente,  $f(j)$  es el largo del mayor prefijo de  $X$  que además es sufijo de  $X$ . Note que  $j = 1$  es un caso especial, puesto que si hay una discrepancia en  $b_1$  el patrón se desliza en una posición.

Si se detecta una discrepancia entre el patrón y el texto cuando se trata de calzar  $b_{j+1}$ , se desliza el patrón de manera que  $b_{f(j)}$  se encuentre donde  $b_j$  se encontraba, y se intenta calzar nuevamente.



Suponiendo que se tiene  $f(j)$  precalculado, la implementación del algoritmo KMP es la siguiente:

```
// n = largo del texto
// m = largo del patron
// Los indices comienzan desde 1

int k=0;
int j=0;
while (k<n && j<m)
{
    while (j>0 && texto[k+1]!=patron[j+1])
    {
        j=f[j];
    }
    if (texto[k+1]==patron[j+1])
    {
        j++;
    }
    k++;
}
// j==m => calce, j el patron estaba en el texto
```

Ejemplo:

Patron = "a a b a a a"  
 1 2 3 4 5 6

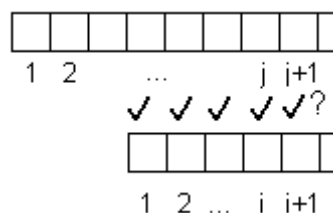
|      |   |   |   |   |   |   |
|------|---|---|---|---|---|---|
| j    | 1 | 2 | 3 | 4 | 5 | 6 |
| f(j) | 0 | 1 | 0 | 1 | 2 | 2 |

Texto: "a a a a b a a b a a b b"  
 j = 0 1 2  
 1 2  
 1 2 3 4 5  
 2 3 4 5 6 → calcel

El tiempo de ejecución de este algoritmo no es difícil de analizar, pero es necesario ser cuidadoso al hacerlo. Dado que se tienen dos ciclos anidados, se puede acotar el tiempo de ejecución por el número de veces que se ejecuta el ciclo externo (menor o igual a  $n$ ) por el número de veces que se ejecuta el ciclo interno (menor o igual a  $m$ ), por lo que la cota es igual a  $O(mn)$ , ¡que es igual a lo que demora el algoritmo de fuerza bruta!

El análisis descrito es pesimista. Note que el número total de veces que el ciclo interior es ejecutado es menor o igual al número de veces que se puede decrementar  $j$ , dado que  $f(j) < j$ . Pero  $j$  comienza desde cero y es siempre mayor o igual que cero, por lo que dicho número es menor o igual al número de veces que  $j$  es incrementado, el cual es menor que  $n$ . Por lo tanto, el tiempo total de ejecución es  $O(n)$ . Por otra parte,  $k$  nunca es decrementado, lo que implica que el algoritmo nunca se devuelve en el texto.

Queda por resolver el problema de definir la función de fracaso,  $f(j)$ . Esto se puede realizar inductivamente. Para empezar,  $f(1)=0$  por definición. Para calcular  $f(j+1)$  suponga que ya se tienen almacenados los valores de  $f(1)$ ,  $f(2)$ , ...,  $f(j)$ . Se desea encontrar un  $i+1$  tal que el  $(i+1)$ -ésimo carácter del patrón sea igual al  $(j+1)$ -ésimo carácter del patrón.



Para esto se debe cumplir que  $i=f(j)$ . Si  $b_{i+1}=b_{j+1}$ , entonces  $f(j+1)=i+1$ . En caso contrario, se reemplaza  $i$  por  $f(i)$  y se verifica nuevamente la condición.

El algoritmo resultante es el siguiente (note que es similar al algoritmo KMP):

```
// m es largo del patron
// los indices comienzan desde 1
int[] f=new int[m];
f[1]=0;
int j=1;
int i;
while (j<m)
```

```

{
  i=f[j];
  while (i>0 && patron[i+1]!=patron[j+1])
  {
    i=f[i];
  }
  if (patron[i+1]==patron[j+1])
  {
    f[j+1]=i+1;
  }
  else
  {
    f[j+1]=0;
  }
  j++;
}

```

El tiempo de ejecución para calcular la función de fracaso puede ser acotado por los incrementos y decrementos de la variable  $i$ , que es  $O(m)$ .

Por lo tanto, el tiempo total de ejecución del algoritmo, incluyendo el preprocesamiento del patrón, es  $O(n+m)$ .

## Algoritmo Boyer-Moore

Hasta el momento, los algoritmos de búsqueda en texto siempre comparan el patrón con el texto de izquierda a derecha. Sin embargo, suponga que la comparación ahora se realiza de derecha a izquierda: si hay una discrepancia en el último carácter del patrón y el carácter del texto no aparece en todo el patrón, entonces éste se puede deslizar  $m$  posiciones sin realizar ninguna comparación extra. En particular, no fue necesario comparar los primeros  $m-1$  caracteres del texto, lo cual indica que podría realizarse una búsqueda en el texto con menos de  $n$  comparaciones; sin embargo, si el carácter discrepante del texto se encuentra dentro del patrón, éste podría desplazarse en un número menor de espacios.

El método descrito es la base del algoritmo Boyer-Moore, del cual se estudiarán dos variantes: Horspool y Sunday.

### Boyer-Moore-Horspool (BMH)

El algoritmo BMH compara el patrón con el texto de derecha a izquierda, y se detiene cuando se encuentra una discrepancia con el texto. Cuando esto sucede, se desliza el patrón de manera que la letra del texto que estaba alineada con  $b_m$ , denominada  $c$ , ahora se alinie con algún  $b_j$ , con  $j < m$ , si dicho calce es posible, o con  $b_0$ , un carácter ficticio a la izquierda de  $b_1$ , en caso contrario (este es el mejor caso del algoritmo).

Para determinar el desplazamiento del patrón se define la *función siguiente* como:

- 0 si  $c$  no pertenece a los primeros  $m-1$  caracteres del patrón (¿Por qué no se considera el carácter  $b_m$ ?).
- $j$  si  $c$  pertenece al patrón, donde  $j < m$  corresponde al mayor índice tal que  $b_j = c$ .

Esta función sólo depende del patrón y se puede precalcular antes de realizar la búsqueda.

El algoritmo de búsqueda es el siguiente:

```
// m es el largo del patron
// los indices comienzan desde 1

int k=m;
int j=m;
while(k<=n && j>=1)
{
    if (texto[k-(m-j)]==patron[j])
    {
        j--;
    }
    else
    {
        k=k+(m-siguiente(a[k]));
        j=m;
    }
}
// j==0 => calce!, j>=0 => no hubo calce.
```

Ejemplo de uso del algoritmo BMH:

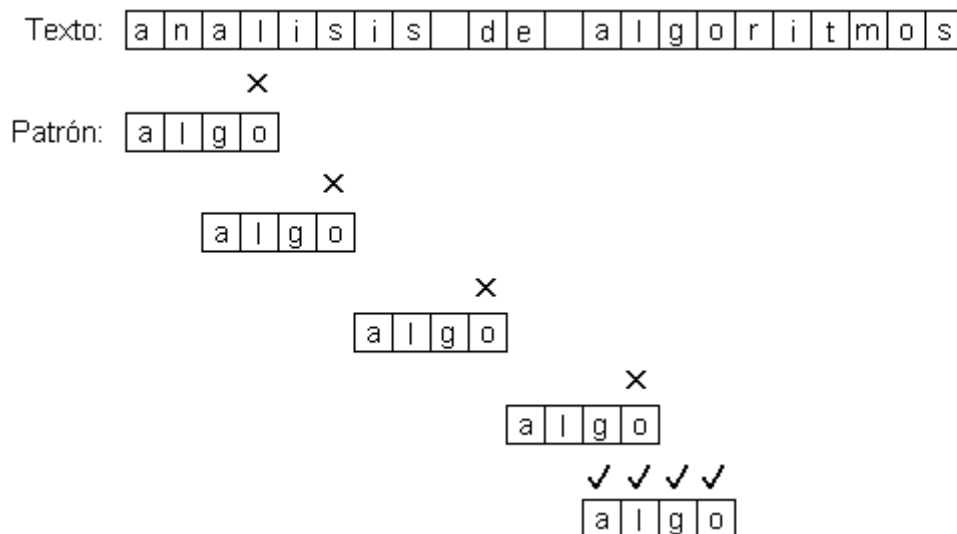


Tabla siguiente:

siguiente(g) = 3  
siguiente(l) = 2  
siguiente(a) = 1

Se puede demostrar que el tiempo promedio que toma el algoritmo BMH es:

$$O\left(n\left(\frac{1}{m} + \frac{1}{2c}\right)\right)$$

donde  $c$  es el tamaño del alfabeto ( $c \ll n$ ). Para un alfabeto razonablemente grande, el

algoritmo es  $O\left(\frac{n}{m}\right)$ .

En el peor caso, BMH tiene el mismo tiempo de ejecución que el algoritmo de fuerza bruta.

## **Boyer-Moore-Sunday (BMS)**

El algoritmo BMH desliza el patrón basado en el símbolo del texto que corresponde a la posición del último carácter del patrón. Este siempre se desliza al menos una posición si se encuentra una discrepancia con el texto.

Es fácil ver que si se utiliza el carácter una posición más adelante en el texto como entrada de la función siguiente el algoritmo también funciona, pero en este caso es necesario considerar el patrón completo al momento de calcular los valores de la función siguiente. Esta variante del algoritmo es conocida como Boyer-Moore-Sunday (BMS).

¿Es posible generalizar el argumento, es decir, se pueden utilizar caracteres más adelante en el texto como entrada de la función siguiente? La respuesta es no, dado que en ese caso puede ocurrir que se salte un calce en el texto.

# Grafos

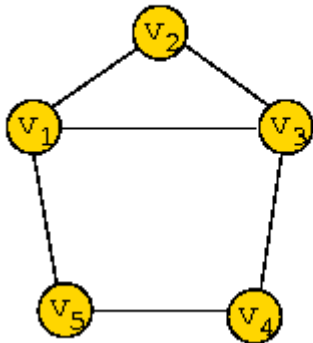
1. [Definiciones Básicas](#)
2. [Recorridos de Grafos](#)
3. [Árbol Cobertor Mínimo](#)
4. [Distancias Mínimas en un Grafo Dirigido](#)

## Definiciones Básicas

Un *grafo* consiste de un conjunto  $V$  de *vértices* (o *nodos*) y un conjunto  $E$  de *arcos* que conectan a esos vértices.

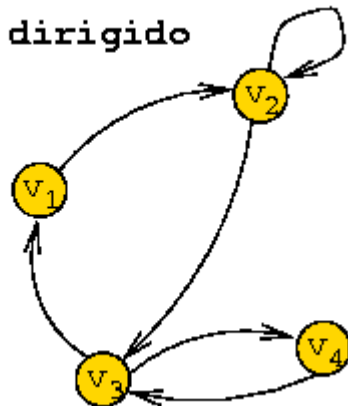
Ejemplos:

**Grafo no dirigido**



$$V = \{v_1, v_2, v_3, v_4, v_5\}$$
$$E = \{ \{v_1, v_2\}, \{v_1, v_3\}, \{v_1, v_5\}, \{v_2, v_3\}, \{v_3, v_4\}, \{v_4, v_5\} \}$$

**Grafo dirigido**



$$V = \{v_1, v_2, v_3, v_4\}$$
$$E = \{ (v_1, v_2), (v_2, v_2), (v_2, v_3), (v_3, v_1), (v_3, v_4), (v_4, v_3), (v_4, v_2) \}$$

Además de esto, los grafos pueden ser extendidos mediante la adición de rótulos (*labels*) a los arcos. Estos rótulos pueden representar costos, longitudes, distancias, pesos, etc.

## Representaciones de grafos en memoria

### *Matriz de adyacencia*

Un grafo se puede representar mediante una matriz  $A$  tal que  $A[i, j] = 1$  si hay un arco que conecta  $v_i$  con  $v_j$ , y 0 si no. La matriz de adyacencia de un grafo no dirigido es simétrica.

Una matriz de adyacencia permite determinar si dos vértices están conectados o no en tiempo constante, pero requieren  $O(n^2)$  bits de memoria. Esto puede ser demasiado para muchos grafos que aparecen en aplicaciones reales, en donde  $|E| \ll n^2$ . Otro problema es que se requiere tiempo  $O(n)$  para encontrar la lista de vecinos de un vértice dado.

### *Listas de adyacencia*

Esta representación consiste en almacenar, para cada nodo, la lista de los nodos adyacentes a él. Para el segundo ejemplo anterior,

```

v1: v2
v2: v2, v3
v3: v1, v4
v4: v3

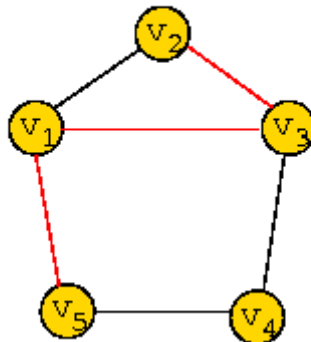
```

Esto utiliza espacio  $O(|E|)$  y permite acceso eficiente a los vecinos, pero no hay acceso al azar a los arcos.

### Camino, ciclos y árboles

Un *camino* es una secuencia de arcos en que el extremo final de cada arco coincide con el extremo inicial del siguiente en la secuencia.

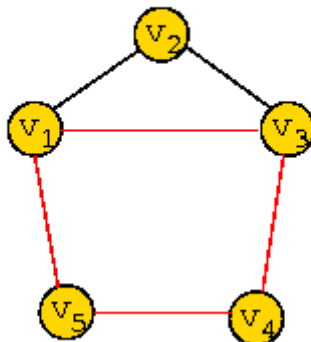
Un camino (en rojo)



Un camino es *simple* si no se repiten vértices, excepto posiblemente el primero y el último.

Un *ciclo* es un camino simple y cerrado.

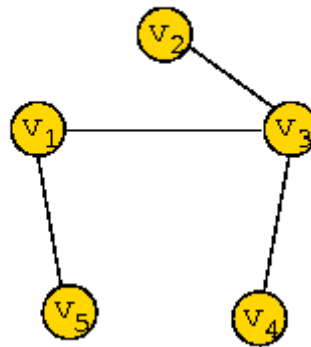
Un ciclo (en rojo)



Un grafo es *conexo* si desde cualquier vértice existe un camino hasta cualquier otro vértice del grafo.

Se dice que un grafo no dirigido es un *árbol* si es conexo y acíclico.

### Un árbol



## Recorridos de Grafos

En muchas aplicaciones es necesario visitar todos los vértices del grafo a partir de un nodo dado. Algunas aplicaciones son:

- Encontrar ciclos
- Encontrar componentes conexas
- Encontrar árboles cobectores

Hay dos enfoque básicos:

- Recorrido (o búsqueda) en profundidad (depth-first search):  
La idea es alejarse lo más posible del nodo inicial (sin repetir nodos), luego devolverse un paso e intentar lo mismo por otro camino.
- Recorrido (o búsqueda) en amplitud (breadth-first search):  
Se visita a todos los vecinos directos del nodo inicial, luego a los vecinos de los vecinos, etc.

### Recorrido en Profundidad (DFS)

A medida que recorremos el grafo, iremos numerando correlativamente los nodos encontrados (1,2,...). Suponemos que todos estos números son cero inicialmente, y utilizamos un contador global  $n$ , también inicializado en cero.

El siguiente algoritmo en pseudo-código muestra cómo se puede hacer este tipo de recorrido recursivamente:

```
DFS(v) // recorre en profundidad a partir del vértice v
{
  ++n;
  DFN[v]=n;
  for(todo w tal que {v,w} está en E y DFN[w]==0)
    DFS(w);
}
```



Para hacer un recorrido en profundidad a partir del nodo  $v$ , utilizamos el siguiente programa principal:

```
n=0;
for (todo w)
    DFN[w]=0;
DFS (v) ;
```

Si hubiera más de una componente conexas, esto no llegaría a todos los nodos. Para esto podemos hacer:

```
n=0;
ncc=0; // número de componentes conexas
for (todo w)
    DFN[w]=0;
while (existe v en V con DFN[v]==0)
{
    ++ncc;
    DFS (v) ;
}
```

Ejercicio: ¿Cómo utilizar este algoritmo para detectar si un grafo es acíclico?

También es posible implementar un recorrido no recursivo utilizando una pila:

```
Pila pila=new Pila();
n=0;
for (todo w)
    DFN[w]=0;
pila.apilar(v); // para recorrer a partir de v
while (!pila.estaVacia())
{
    v=pila.desapilar();
    if (DFN[v]==0) // primera vez que se visita este nodo
    {
        ++n;
        DFN[v]=n;
        for (todo w tal que {v,w} esta en E)
            pila.apilar(w);
    }
}
```

## Recorrido en amplitud (BFS)

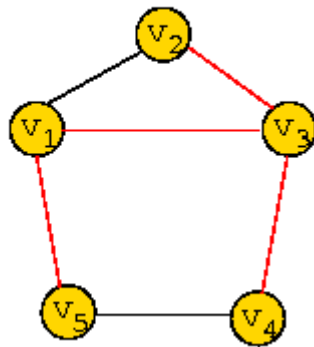
La implementación es similar a la de DFS, pero se utiliza una *cola* en lugar de una pila.

El resultado es que los nodos se visitan en orden creciente en relación a su distancia al nodo origen.

## Árboles cobertores

Dado un grafo  $G$  no dirigido, conexo, se dice que un subgrafo  $T$  de  $G$  es un árbol cobertor si es un árbol y contiene el mismo conjunto de nodos que  $G$ .

### Un árbol cobertor (en rojo)



Todo recorrido de un grafo conexo genera un árbol cobertor, consistente del conjunto de los arcos utilizados para llegar por primera vez a cada nodo.

Para un grafo dado pueden existir muchos árboles cobertores. Si introducimos un concepto de "peso" (o "costo") sobre los arcos, es interesante tratar de encontrar un árbol cobertor que tenga costo mínimo.

### Árbol Cobertor Mínimo

En esta sección veremos dos algoritmos para encontrar un árbol cobertor mínimo para un grafo no dirigido dado, conexo y con costos asociados a los arcos. El costo de un árbol es la suma de los costos de sus arcos.

#### Algoritmo de Kruskal

Este es un algoritmo del tipo "avaro" ("*greedy*"). Comienza inicialmente con un grafo que contiene sólo los nodos del grafo original, sin arcos. Luego, en cada iteración, se agrega al grafo el arco más barato que no introduzca un ciclo. El proceso termina cuando el grafo está completamente conectado.

En general, la estrategia "avara" no garantiza que se encuentre un óptimo global, porque es un método "miope", que sólo optimiza las decisiones de corto plazo. Por otra parte, a menudo este tipo de métodos proveen buenas heurísticas, que se acercan al óptimo global.

En este caso, afortunadamente, se puede demostrar que el método "avaro" logra siempre encontrar el óptimo global, por lo cual un árbol cobertor encontrado por esta vía está garantizado que es un árbol cobertor mínimo.

Una forma de ver este algoritmo es diciendo que al principio cada nodo constituye su propia componente conexa, aislado de todos los demás nodos. Durante el proceso de construcción del árbol, se agrega un arco sólo si sus dos extremos se encuentran en componentes conexas distintas, y luego de agregarlo esas dos componentes conexas se fusionan en una sola.

Para la operatoria con componentes conexas supondremos que cada componente conexa se identifica mediante un *representante canónico* (el "líder" del conjunto), y que se dispone de las siguientes operaciones:

$\text{Union}(a,b)$ : Se fusionan las componentes canónicas representadas por  $a$  y  $b$ , respectivamente.

$\text{Find}(x)$ : Encuentra al representante canónico de la componente conexa a la cual pertenece  $x$ .

Con estas operaciones, el algoritmo de Kruskal se puede escribir así:

```

Ordenar los arcos de E en orden creciente de costo;
C = { {v} | v está en V }; // El conjunto de las componentes
conexas
while( |C| > 1 )
{
    Sea e={v,w} el siguiente arco en orden de costo creciente;
    if (Find(v) != Find(w))
    {
        Agregar e al árbol;
        Union(Find(v), Find(w));
    }
}

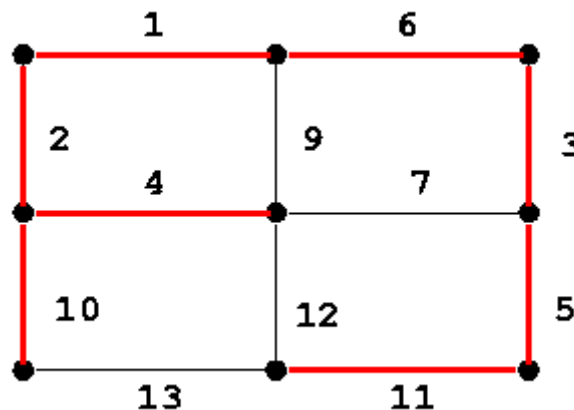
```

El tiempo que demora este algoritmo está dominado por lo que demora la ordenación de los arcos. Si  $|V|=n$  y  $|E|=m$ , el tiempo es  $O(m \log m)$  más lo que demora realizar  $m$  operaciones  $\text{Find}$  más  $n$  operaciones  $\text{Union}$ .

Es posible implementar Union-Find de modo que las operaciones  $\text{Union}$  demoran tiempo constante, y las operaciones  $\text{Find}$  un tiempo *casi* constante. Más precisamente, el costo amortizado de un  $\text{Find}$  está acotado por  $\log^* n$ , donde  $\log^* n$  es una función definida como el número de veces que es necesario tomar el logaritmo de un número para que el resultado sea menor que 1.

Por lo tanto, el costo total es  $O(m \log m)$  o, lo que es lo mismo,  $O(m \log n)$  (¿por qué?).

Ejemplo:



La correctitud del algoritmo de Kruskal viene del siguiente lema:

### Lema

Sea  $v'$  subconjunto propio de  $v$ , y sea  $e=\{v,w\}$  un arco de costo mínimo tal que  $v$  está en  $v'$  y  $w$  está en  $v-v'$ . Entonces existe un árbol cobertor mínimo que incluye a  $e$ .

Este lema permite muchas estrategias distintas para escoger los arcos del árbol. Veamos por ejemplo la siguiente:

### Algoritmo de Prim

Comenzamos con el arco más barato, y marcamos sus dos extremos como "alcanzables". Luego, a cada paso, intentamos extender nuestro conjunto de nodos alcanzables agregando el arco más barato que tenga uno de sus extremos dentro del conjunto alcanzable y el otro fuera de él.

De esta manera, el conjunto alcanzable se va extendiendo como una "mancha de aceite".

```
Sea  $e=\{v,w\}$  un arco de costo mínimo en  $E$ ;  
Agregar  $e$  al árbol;  
 $A=\{v,w\}$ ; // conjunto alcanzable  
while ( $A \neq V$ )  
{  
    Encontrar el arco  $e=\{v,w\}$  más barato con  $v$  en  $A$  y  $w$  en  $V-A$ ;  
    Agregar  $e$  al árbol;  
    Agregar  $w$  al conjunto alcanzable  $A$ ;  
}
```

Para implementar este algoritmo eficientemente, podemos mantener una tabla donde, para cada nodo de  $V-A$ , almacenamos el costo del arco más barato que lo conecta al conjunto  $A$ . Estos costos pueden cambiar en cada iteración.

Si se organiza la tabla como una cola de prioridad, el tiempo total es  $O(m \log n)$ . Si se deja la tabla desordenada y se busca linealmente en cada iteración, el costo es  $O(n^2)$ . Esto último es mejor que lo anterior si el grafo es denso.

### Distancias Mínimas en un Grafo Dirigido

En este problema los rótulos de los arcos se interpretan como distancias. La distancia (o largo) de un camino es la suma de los largos de los arcos que lo componen.

El problema de encontrar los caminos más cortos tiene dos variantes:

- Caminos más cortos desde un nodo "origen"  
Encontrar los  $n$  caminos más cortos desde un nodo dado  $s$  hasta todos los nodos del grafo.
- Todos los caminos más cortos  
Encontrar los  $n^2$  caminos más cortos entre todos los pares de nodos (origen, destino). Esto se puede resolver resolviendo  $n$  veces el problema anterior,

cambiando cada vez de nodo origen, pero puede ser más eficiente encontrar todos los caminos de una sola vez.

### Algoritmo de Dijkstra para los caminos más cortos desde un nodo "origen"

La idea del algoritmo es mantener un conjunto  $A$  de nodos "alcanzables" desde el nodo origen e ir extendiendo este conjunto en cada iteración.

Los nodos alcanzables son aquellos para los cuales ya se ha encontrado su camino óptimo desde el nodo origen. Para esos nodos su distancia óptima al origen es conocida. Inicialmente  $A = \{s\}$ .

Para los nodos que no están en  $A$  se puede conocer el camino óptimo desde  $s$  *que pasa sólo por nodos de  $A$* . Esto es, caminos en que todos los nodos intermedios son nodos de  $A$ . Llamemos a esto su camino óptimo tentativo.

En cada iteración, el algoritmo encuentra el nodo que no está en  $A$  y cuyo camino óptimo tentativo tiene largo mínimo. Este nodo se agrega a  $A$  y su camino óptimo tentativo se convierte en su camino óptimo. Luego se actualizan los caminos óptimos tentativos para los demás nodos.

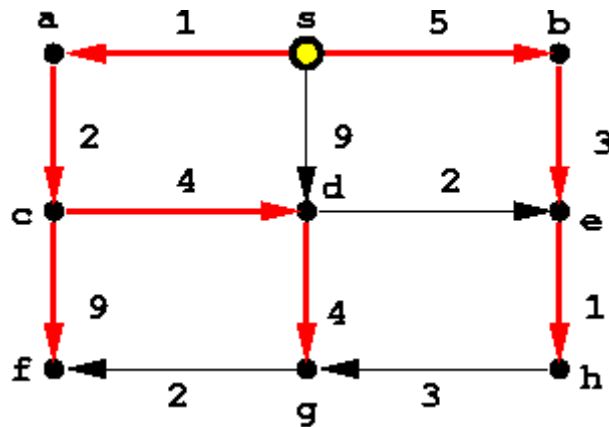
El algoritmo es el siguiente:

```
A={s};
D[s]=0;
D[v]=cost(s,v) para todo v en V-A; // infinito si el arco no
existe
while (A!=V)
{
    Encontrar v en V-A tal que D[v] es mínimo;
    Agregar v a A;
    for(todo w tal que (v,w) está en E)
        D[w]=min(D[w], D[v]+cost(v,w));
}
```

#### Implementaciones:

- Usando una cola de prioridad para la tabla  $D$  el tiempo es  $O(m \log n)$ .
- Usando un arreglo con búsqueda secuencial del mínimo el tiempo es  $O(n^2)$ .

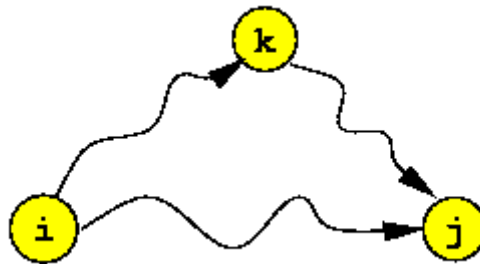
#### Ejemplo:



### Algoritmo de Floyd para todas las distancias más cortas

Para aplicar este algoritmo, los nodos se numeran arbitrariamente  $1, 2, \dots, n$ . Al comenzar la iteración  $k$ -ésima se supone que una matriz  $D[i, j]$  contiene la distancia mínima entre  $i$  y  $j$  medida a través de caminos que pasen sólo por nodos intermedios de numeración  $< k$ .

Estas distancias se comparan con las que se obtendrían si se pasara una vez por el nodo  $k$ , y si de esta manera se obtendría un camino más corto entonces se prefiere este nuevo camino, de lo contrario nos quedamos con el nodo antiguo.



Al terminar esta iteración, las distancias calculadas ahora incluyen la posibilidad de pasar por nodos intermedios de numeración  $\leq k$ , con lo cual estamos listos para ir a la iteración siguiente.

Para inicializar la matriz de distancias, se utilizan las distancias obtenidas a través de un arco directo entre los pares de nodos (o infinito si no existe tal arco). La distancia inicial entre un nodo y sí mismo es cero.

```
for (1 ≤ i, j ≤ n)
    D[i, j] = cost(i, j); // infinito si no hay arco entre i y j
for (1 ≤ i ≤ n)
    D[i, i] = 0;

for (k = 1, ..., n)
    for (1 ≤ i, j ≤ n)
        D[i, j] = min(D[i, j], D[i, k] + D[k, j]);
```

El tiempo total que demora este algoritmo es  $O(n^3)$ .

## Algoritmo de Warshall para cerradura transitiva

Dada la matriz de adyacencia de un grafo (con  $D[i, j]=1$  si hay un arco entre  $i$  y  $j$ , y 0 si no), resulta útil calcular la matriz de conectividad, en que el casillero respectivo contiene un 1 si y sólo si existe un *camino* (de largo  $\geq 0$ ) entre  $i$  y  $j$ . Esta matriz se llama también la *cerradura transitiva* de la matriz original.

La matriz de conectividad se puede construir de la siguiente manera:

1. Primero calcular la matriz de distancias mínimas usando Floyd.
2. Luego sustituir cada valor infinito por 0 y cada valor no infinito por 1.

Mejor todavía, podríamos modificar el algoritmo de Floyd para que vaya calculando con ceros y unos directamente, usando las correspondencias siguientes:

|           |           |           |    |   |   |
|-----------|-----------|-----------|----|---|---|
| min       | infinito  | <infinito | or | 0 | 1 |
| infinito  | infinito  | <infinito | 0  | 0 | 1 |
| <infinito | <infinito | <infinito | 1  | 1 | 1 |

|           |          |           |     |   |   |
|-----------|----------|-----------|-----|---|---|
| +         | infinito | <infinito | and | 0 | 1 |
| infinito  | infinito | infinito  | 0   | 0 | 0 |
| <infinito | infinito | <infinito | 1   | 0 | 1 |

El resultado se denomina Algoritmo de Warshall:

```
for (1<=i, j<=n)
    D[i, j]=adyacente(i, j); // 1 si existe, 0 si no
for (1<=i<=n)
    D[i, i]=1;

for (k=1, ..., n)
    for (1<=i, j<=n)
        D[i, j]=D[i, j] or (D[i, k] and D[k, j]);
```

# Algoritmos Probabilísticos

En muchos casos, al introducir elecciones aleatorias en un algoritmo se pueden obtener mejores rendimientos que al aplicar el algoritmo determinístico puro.

Un *algoritmo tipo Montecarlo* asegura un tiempo fijo de ejecución, pero no está garantizado que la respuesta sea correcta, aunque lo puede ser con alta probabilidad.

Un *algoritmo tipo Las Vegas* siempre entrega la respuesta correcta, pero no garantiza el tiempo total de ejecución, aunque con alta probabilidad éste será bajo.

**Ejemplo: algoritmo tipo Montecarlo para verificar la multiplicación de dos matrices.**

Sean  $A, B, C$  matrices de  $N \times N$ . Se desea chequear si  $A \cdot B = C$ . Determinísticamente, esto toma tiempo  $O(n^3)$  usando el algoritmo estándar.

Probabilísticamente, se puede chequear si  $A \cdot B = C$  en tiempo  $O(n^2)$  con baja probabilidad de error. El algoritmo es el siguiente:

```
for (i=1; i<=k; i++)
{
  Generar un vector aleatorio de  $N \times 1$  con entradas en  $\{-1, 1\}$ ;
  if ( $A \cdot (Bx) \neq Cx$ )
    return false; //  $A \cdot B \neq C$ 
}
return true; //  $A \cdot B = C$  con alta probabilidad
```

La complejidad de este algoritmo es  $\Theta(n^2)$ . A continuación se muestra que para este algoritmo:

$$\Pr(\text{error}) \leq \frac{1}{2^k}$$

Sea  $D = A \cdot B$ . Si  $D[i, j] \neq C[i, j]$  para algún  $i, j$ , entonces  $Dx = Cx \Rightarrow Dx' \neq Cx'$ , donde  $x'$  es el vector donde se cambia  $x_j$  por  $-x_j$ . Por lo tanto, en cada iteración del algoritmo se tiene que  $\Pr(\text{error}) \leq 1/2$ , y dado que se realizan  $k$  iteraciones se tiene que  $\Pr(\text{error}) \leq 1/2^k$ .

**Ejemplo: algoritmo tipo Las Vegas para colorear conjuntos.**

Sean  $k$  conjuntos  $C_1, \dots, C_k$  con  $r$  elementos cada uno (**no disjuntos**), y  $k \leq 2^{r-2}$ . Se pide colorear los elementos de color *rojo* o *azul*, tal que ningún  $C_i$  sea homogéneo.

Algoritmo:



```

while (true)
{
    Colorear los elementos aleatoriamente;
    if (ningún  $C_i$  es homogéneo)
        break;
}

```

¿Cuántas veces es necesario repetir el ciclo para obtener una respuesta?

$$Pr(C_i \text{ homogéneo}) = Pr(\text{todos los elementos de } C_i \text{ rojos}) + Pr(\text{todos los elementos de } C_i \text{ azules}) = 1/2^r + 1/2^r = 1/2^{r-1}$$

$$\Rightarrow Pr(\text{algún } C_i \text{ homogéneo}) = k/2^{r-1} \leq 1/2 \text{ (ya que } k \leq 2^{r-2}).$$

Esto implica que en promedio el ciclo se ejecuta 2 veces  $\Rightarrow O(k \cdot r)$ .